CONVEX C
Programmer's Reference

Document No. 720-001530-001

Second Edition
May 1990

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX C Programmer's Reference
Order No. DSW-048
Second Edition

Printed in the United States of America

CONVEX C V4.0

The CONVEX C compiler translates a program written in C into an object module that can be combined with library routines or other object modules and executed on a CONVEX computer.

CONVEX C documentation:

CONVEX ANSI C Concepts
CONVEX C User's Guide
CONVEX C Language Reference Manual
CONVEX C Optimization Guide
CONVEX C Quick Reference
CONVEX C Master Index
CONVEX C Programmer's Reference

The man pages contained in this programmer's reference include:

abort(3)
abs(3)
accept(2)
access(2)
acct(2)
adjtime(2)
alarm(3c)
asiostat(2)
assert(3x)
atof(3)
bind(2)
brk(2)
bstring(3)
cc(1)
cfgetospeed(3)
chdir(2)
chmod(2)
chown(2)
chroot(2)
clock(3)
close(2)
closeshares(3)
connect(2)
cpp(1)
creat(2)
crypt(3)
ctermid(3)
ctime(3)
ctype(3)
curses(3x)
cuserid(3)
cvxprusage(2)
dbm(3x)
difftime(3)
directory(3)
dup(2)
ecvt(3)
end(3)
erf(3m)
errno.h(3)
execl(3)
execve(2)
exit(2)
exit(3)
exp(3m)
exportent(3)
exportfs(2)
faillog(2)
fchmod(2)
fchown(2)
fclose(3s)
fcntl(2)
ferror(3s)
float.h(3)
flock(2)
floor(3m)
fopen(3s)
fork(2)
fread(3s)
frexp(3)
fseek(3s)
fsync(2)
ftime(3c)
gamma(3m)
getactent(3)
getacwent(3)
getaid(2)
getc(3s)
getcwd(3)
getdirentries(2)
getdiskbyname(3x)
getdomainname(2)
getdtablesize(2)
getenv(3)
getfh(2)
getfpmode(3)
getfsent(3x)
getgid(2)
getgrent(3)
getgrgid(3)
getgroups(2)
gethostid(2)
gethostname(2)
getitimer(2)
getlogin(3)
getmntent(3)
getopt(3)
getpagesize(2)
getpass(3)
getpattr(2)
getpeername(2)
getpflags(2)
getpgrp(2)
getpid(2)
getpriority(2)
getpty(3)
getpw(3c)
getpwent(3)
getpwrestent(3)
getpwuid(3)
getrlimit(2)
getrusage(2)
gets(3s)
getshares(3)
getshput(3)

getsockname(2)
getsockopt(2)
getsysinfo(2)
gettimeofday(2)
getttyent(3)
getuid(2)
getusershell(3)
getwd(3)
hypot(3m)
initgroups(3x)
insque(3)
intro(2)
intro(3)
intro(3c)
intro(3m)
intro(3s)
intro(3x)
ioctl(2)
j0(3m)
kill(2)
killpg(2)
limits.h(3)
limits(2)
link(2)
listen(2)
lockf(3)
lseek(2)
lstat(2)
malloc(3)
mbstowcs(3)
mkdir(2)
mkfifo(3)
mknod(2)
mktemp(3)
mmap(2)
monitor(3)
mount(2)
mremap(2)
mset(3)
msleep(2)
msync(2)
munmap(2)
ndbm(3)
nfabort(3)
nfcomment(3)
nfssvc(2)
nice(3c)
nlist(3)
open(2)
openshares(3)
pathconf(3)
pattach(2)
pause(3c)
perror(3)
pgetregid(2)
pipe(2)
plot(3x)
popen(3)
printf(3s)
profil(2)
psetregid(2)
psignal(3)
putc(3s)
puts(3s)
putshares(3)
qsort(3)
quotactl(2)
rand(3c)
random(3)
rcmd(3x)
rcvtir(3m)
read(2)
readlink(2)
readv(2)
reboot(2)
recv(2)
regex(3)
rename(2)
resolver(3)
rexec(3x)
rmdir(2)
scandir(3)
scanf(3s)
select(2)
send(2)
setaid(2)
setbuf(3s)
setfpmode(3)
setgroups(2)
setjmp(3)
setlimits(3)
setlocale(3)
setpgid(2)
setpgrp(2)
setregid(2)
setreuid(2)
setsid(2)
setuid(3)
setupshares(3)
sharesfile(3)
shutdown(2)
sigaction(2)
sigblock(2)
signal(3c)
sigpause(2)
sigpending(2)
sigprocmask(3)
sigsetjmp(3)
sigsetmask(2)
sigsetops(3)
sigstack(2)
sigsuspend(3)
sigvec(2)
sin(3m)
sinh(3m)
sleep(3)
socket(2)
socketpair(2)
stat(2)
statfs(2)
stdarg(3)
stdarg.h(3)
stddef.h(3)
string(3)
stringcat(3)
stringcmp(3)
stringcpy(3)
stringsearch(3)
strtod(3)
stty(3c)
swab(3)
swapon(2)
symlink(2)
sync(2)
syscall(2)
sysconf(3)
syslog(3)
system(3)
tape(3)
tas(3)
tcgetattr(3)
tcgetpgrp(3)
tcsendbreak(3)
termcap(3x)
time(3c)
times(3c)
tmpfile(3s)
truncate(2)
ttyname(3)
tzset(3)
umask(2)
uname(2)
ungetc(3s)
unlink(2)
unmount(2)
utime(3c)
utimes(2)
vadvise(2)
valloc(3)
varargs(3)
vc(1)
vcpp(1)
vfork(2)
vhangup(2)
vlimit(3c)
vprintf(3s)
vtimes(3c)
wait(2)
write(2)
writev(2)

NAME

cc – CONVEX C compiler

SYNOPSIS

cc [*option* ...] *file* ... [*loader_option* ...]

DESCRIPTION

cc is the CONVEX C compiler. It accepts several types of arguments:

Filenames whose names end with ".c" or ".C" must be C source programs. They are compiled and each object program is placed in a file in the current directory with the same root name as the source filename and a suffix of ".o". For example, compiling /x/y/z.c will place the object code in ./z.o.

Filenames which end with ".s" must be assembly source programs and are assembled, producing a ".o" file.

Other files may be object files or libraries; they are passed to *ld* to be linked together with the objects from all compilations and assemblies. If a single source file is compiled and loaded by a *cc* command, the ".o" file is deleted.

COMPATIBILITY

The CONVEX C compiler operates in one of four compatibility modes.

–ext This mode provides an extended implementation of ANSI C and an ANSI C and POSIX compatible library system. This is the default.

–std This option causes the compiler to operate as an ANSI C conforming compiler. Certain extensions, notably the long long type, are not available in this mode. When linking occurs in this mode an ANSI C and POSIX P1003.1 conforming library system is used; many of the extensions provided by the default library system are not available.

–str This option causes the compiler to operate as an ANSI C conforming compiler and to attempt to detect features of the program which cause it not to conform to ANSI Standard C. When linking occurs in this mode, only the functions defined by ANSI Standard C are available.

–pcc Forces language and library interpretation based on the the original Kernighan and Ritchie definition, the Common C Compiler, and traditional Unix systems. This mode is compatible with CONVEX C V3.0 and other C compilers previously shipped by CONVEX.

These options are interpreted before all others, regardless of their position on the command line.

OPTIMIZATION OPTIONS

–alias standard

–alias cautious

–alias worst

The -alias option affects the assumptions the compiler makes regarding overlapping of variables, particularly when referenced by a pointer. *standard* causes the compiler to assume the program obeys the ANSI C standard. *worst* causes the compiler to do worst case alias analysis; it assumes that a pointer may reference any piece of memory other than an *auto* variable whose address has never been taken. *cautious* causes the compiler to behave as it does for *standard* except when casts between pointer types are seen. In that case it behaves like *worst*.

Incorrect code may be generated if the implied conditions do not hold.

When the *-pcc* flag is provided, *worst* is the default, otherwise, the default is *cautious*.

–alias array_args
Assume that formal array parameters do not overlap each other or any external variable that is used in the function (unless all uses are read-only). **Incorrect code will be generated if this assumption is not true.** This conflicts with the language definition, but may allow greater optimization, particularly vectorization, to occur.

This option may not be used if the formal parameter itself is assigned by the function (e.g., formalParameter = &x[10]); assignments may be made to the elements of the formal (e.g., formalParameter[10] = x[10]).

–alias ptr_args
Assume that the variables identified by formal pointer parameters do not overlap each other or any external variable that is used in the function (unless all uses are read-only). **Incorrect code will be generated if this assumption is not true.** This conflicts with the language definition, but may allow greater optimization, particularly vectorization, to occur.

This option may not be used if the formal parameter itself is assigned by the function (e.g., formalParameter = &x[10]); assignments may be made to the elements of the formal (e.g., formalParameter[10] = x[10]).

–ds Causes the compiler to perform dynamic selection on loops selected by the compiler on the basis of profitability. Multiple copies of the loop running sequential, vector or parallel modes are generated; the optimal version gets executed based on the trip count of the loop. This option is only effective at optimization level *-O2* and *-O3*.

–ep n Specifies the expected number of processors that will execute the program. Optimizes single-program performance at the expense of overall system throughput. n must be in the range 1 through 4.

–no No machine-independent optimization is performed; this is the default but can be overridden by the *-O* options.

–O*n* Performs machine-independent optimizations, level *n*, where:
n=0 means basic block scalar optimization;
n=1 means *-O0* function-wide scalar optimization;
n=2 means *-O1* plus vectorization;
n=3 means *-O2* plus parallelization.
If this option is not specified, the compiler performs no machine independent optimization.

–rl Causes the compiler to perform loop replication optimizations (loop unrolling, dynamic code selection) on loops selected by the compiler on the basis of profitability. This option is equivalent to *-ds -ur*. This option is only effective with options *-O2* and *-O3*.

–uo Performs potentially unsafe optimizations, i.e., may move the evaluation of common subexpressions and/or invariant code from within conditionally executed code. The code is then executed unconditionally.

–ur Causes the compiler to perform loop unrolling on loops selected by the compiler on the basis of profitability. This option is only effective at optimization levels *-O2* and *-O3*.

–va A synonym for *-alias array_args*.

CODE GENERATION OPTIONS

–asm This flag is silently ignored; it is no longer required.

–c Do not run the linker. The object module generated from *x.c* or *x.s* is left in *x.o*.

–extern distinct

–extern same

The *-extern* option may be used to control the interpretation of a program in which different files declare an uninitialized variable (e.g., `"int i;"`) at file scope. When *same* is used both files refer to the same variable. This is the default in all modes, is traditional on BSD Unix Systems and is a conforming extension to Standard C. When *distinct* is used the files refer to distinct variables; this results in a multiply defined symbol being detected by the linker. This is the definition provided by the ANSI C standard.

–fd A synonym for *-float sp_ops*, described below.

–fi Translate floating-point constants to IEEE format and perform floating-point operations in IEEE mode. This option requires that the machine be equipped with IEEE floating-point hardware. If no floating-point format is specified, the site default is used. Arithmetic performed under this option does not conform to the IEEE standard.

–float sp_ops

–float dp_ops

–float sp_const

–float dp_const

These options control the floating-point system used by the compiler. *sp_ops* and *dp_ops* control the precision of the operations performed on float operands. When *sp_ops* is specified, 32 bit operations are performed. When *dp_ops* is specified, the float operands are converted to double and 64 bit operations are performed. *sp_ops* generally causes programs using float variables to run faster but some accuracy may be lost.

sp_const and *dp_const* control the representation of unsuffixed floating-point constants (which are normally represented in double precision). *sp_const* causes these constants to be represented in single precision. Some loss of accuracy may result. *dp_const* explicitly invokes the default.

Use of *sp_ops* is more effective when combined with *sp_const* since expressions like $x == 0.0$ are performed in double precision when only *sp_ops* is specified.

sp_ops is the default in all but *-pcc* mode.

dp_const is the default in all modes.

–fn Translate floating-point constants to native CONVEX format and perform floating-point operations in native mode. If no floating-point format is specified, the site default is used.

–parens ignore

–parens explicit

–parens implicit

Toggle the manner in which parentheses and the ANSI C grammar are used to determine the order of evaluation of expressions.

The value *ignore* gives the compiler freedom to re-order expressions; even explicit parentheses are ignored. This is the default in *-pcc* and *-ext* modes and is the traditional

method used in C.

The value *explicit* tells the compiler to honor explicit parentheses but ignore ordering implied by the grammar and associativity rules. This is similar to the rules used by Fortran and many other languages.

The value *implicit* tells the compiler that explicit parentheses and ordering implied by the grammar and associativity rules should be honored. This means that a+b+c must be evaluated in the order implied by (a+b)+c. This is the default in *-std* and *-str* modes.

These rules apply only to floating-point expressions; operations on other types are always subject to re-ordering.

–re Specifies that the compiler generate re-entrant code by generating both a scalar and a parallel version of parallel loops. The default, at optimization level *-O3*, is to generate non-reentrant code for functions with parallel regions. This option has no effect on functions without parallel code; it should be used to compile a function for which the compiler generates parallel code, if you wish to call that function from a parallel region. It is always **safe** to use *-re*; the text segment will be unnecessarily large if *-re* is used when it is not required.

–S Generates symbolic assembly code for each program unit in a source file. Assembler output for a source file *x.c* is put on file *x.s*; no object file is generated. The default is to write only an object file.

–sc Instructs the compiler to check the program for compilation errors. No optimization, vectorization, or code generation is performed.

–string read_write
String constants are stored in the data segment and may be modified. This is the default when *-pcc* is specified.

–string read_only
String constants are stored in the text segment and may not be modified. This option increases the sharability of a program when multiple copies of it are running at once. This is the default if *-pcc* is not specified.

–tm x Specifies the target machine architecture. *x* may be c1, C1, c2, or C2; the default value is the type of machine the compiler is running on. If c2 or C2 is used, the resulting code will not run on a C1. If c1 or C1 is used, the resulting code will run on either a C1 or a C2 but may run less quickly on a C2 than it would if compiled with *-tm* C2. When invoking the linker machine dependent versions of some libraries are used.

DEBUGGING AND PROFILING OPTIONS

–db Produces additional information for use by the symbolic debugger, *csd*, and the post-mortem dump utility (*pmd*). It also passes the **-lg** flag to the loader. This flag can be used with all levels of optimization, but unless the **-no** option (NO optimization) is specified, there may be source statements for which no debugging information is generated for *csd*. Information about variables declared in the inner blocks of functions is produced only at level *-no*.

–p Causes the compiler to produce code that counts the number of times each routine is called. If loading takes place, replaces the standard startup routine by one that automatically calls *monitor*(3) at the start and arranges to write out a "mon.out" file at normal termination of execution of the object program. A profiled library is searched, instead of the standard C library. An execution profile can then be generated by use of *prof*(1)

(optional product).

-pa Produces counting code for routine-level and loop-level profiles using the CXpa utility. Refer to the *CONVEX Performance Analyzer (CXpa) User's Guide* for more information. CXpa is an optional product.

-pab Produces counting code for block-level profiles using the CXpa utility. Refer to the *CONVEX Performance Analyzer (CXpa) User's Guide* for more information. CXpa is an optional product.

-par Includes instrumentation for routine, loop, and region level profiles using the CXpa utility. Refer to the *CONVEX Performance Analyzer (CXpa) User's Guide* for more information. CXpa is an optional product.

-pb Causes the compiler to produce statement-level counting code that produces an execution profile named *bmon.out* at normal termination. Listings of source-level execution counts can then be obtained using *bprof*(1). *bprof* is an optional product.

-pg Causes the compiler to produce counting code in the manner of **-p**, but invokes a runtime recording mechanism that keeps more extensive statistics and produces a *gmon.out* file at normal termination. An execution profile can then be generated by use of *gprof*(1) (optional product).

COMPILER OUTPUT OPTIONS

-d *name*

-d *name=e*

-d *name=w*

The *-d* option provides low level control of diagnostic output. The form *-d name* suppresses the named message. The form *-d name=w* causes the named message to be reported as a warning. The form *-d name=e* causes the named message to be reported as an error. The message names are documented in the CONVEX C User's Guide.

-nv Suppresses all vectorization summary messages (synonymous with *-or none*).

-nw Suppresses all warning diagnostic messages.

-or *table*

Specifies the contents of the optimization report to be produced; either the loop table, the array table, or both can be displayed. The value for *table* can be *none, all, array* or *loop*. If this option is not specified, only the loop table is displayed.

MISCELLANEOUS OPTIONS

-B*dir* Finds substitute compiler (*cocc, cpp,* and *errmsgc*) in the directory *dir*. If *dir* is not specified the standard backup version in the directory */usr/lib/oldcc* is used instead. For example, the command `cc -B/usr/new` would invoke */usr/new/cocc* instead of the default */usr/lib/cc/cocc*.

-o*name*

Specifies that *name* is the name of the executable file produced by *ld*. If this option is not specified, the default name is *a.out*. If **-c** was specified and there is only one file to compile or assemble, *name* is the name of the object module produced.

-tl *time*

Sets the maximum CPU time limit on compilations to *time* minutes. If the CPU time exceeds the allotted time, the compilation is aborted.

–vn Identifies compiler version. Outputs the version numbers of *cc*, *cpp*, *cocc*, and *errmsgc* to *stderr*.

UNIX COMPATIBILITY OPTIONS

The following options are provided for compatibility with compilers of other vendors.

–g A synonym for *-db*.

–n Ignored with a warning.

–O A synonym for *-O1*.

–OL A synonym for *-O1*.

–V A synonym for *-vn*.

–w A synonym for *-nw*

PREPROCESSOR OPTIONS

The CONVEX C compiler invokes the C macro preprocessor *cpp* on each file and compiles the resulting text. The following options affect the preprocessing phase of translation.

–C

–D*name*

–D*name*=*def*

–I*dir*

–U*name*

These options are passed directly to *cpp(1)* without interpretation by *cc*. See *cpp(1)* for further details.

–E Run *cpp* on the named C source files and send the result to the standard output stream (*stdout*). The source is not checked for errors and no object, assembly or executable code is produced.

–k Run *cpp* on the named C source files and generate *make* dependency descriptions on the standard output stream (*stdout*).

If the compiler option *-pcc* is present it is passed to each invocation of the preprocessor.

PREDEFINED SYMBOLS

The following macros are predefined when *cc* invokes *cpp*:

_CONVEX_SOURCE

_POSIX_SOURCE

The symbols _CONVEX_SOURCE and _POSIX_SOURCE are predefined in extended mode. In *-std* mode the user will generally want to define the symbol _POSIX_SOURCE to make the POSIX symbols available in the include files.

_CONVEX_FLOAT_

_IEEE_FLOAT_

The symbol _CONVEX_FLOAT_ is defined when the compiler is operating in native float point mode, _IEEE_FLOAT_ is defined in IEEE mode. See the description of the options *-fi* and *-fn* above.

__STDC__
The symbol __STDC__ is defined when either the *-std* or *-str* flags are specified. The definition of this symbol indicates that the compiler conforms to the ANSI C Standard. This definition may not be removed by the *-U* option or *#undef*.

__stdc__
The symbol __stdc__ is defined in all modes except the *-pcc* mode. It indicates that the compiler is an ANSI style compiler, but not that it is conforming. The conforming modes define both __stdc__ and __STDC__.

__NO_INLINE
The symbol __NO_INLINE is defined in *-str* and *-std* modes to suppress recognition of macros which define certain library functions. This is necessary to get conforming implementations of some functions which would otherwise not set *errno* in a standard conforming manner. See the *math.h* man page for more details.

__convexc__
This symbol is always defined. It should be used to identify the compiler. Other symbols defined by the preprocessor (see *cpp(1)*) identify the machine and operating system.

convexvc
This symbol is defined in *-pcc* mode. It's use is obsolescent; the symbol __convexc__ should be used instead.

The preprocessor defines a number of symbols as well; see *cpp(1)* for a discussion of these symbols.

Other names beginning with "__" or _[A-Z] may be predefined by *cc*. Such names are reserved to CONVEX; their usage or availability may change in subsequent releases. Applications should not depend on the *presence or absence* of such names except as defined above.

LOADER USAGE

A number of compiler options impact the manner in which *cc* invokes the loader. Options for the loader may also be specified directly by the user. CONVEX recommends always using the appropriate compiler to invoke the loader rather than invoking it directly. This will insulate the program from changes in library structure when new compiler releases are installed.

The compiler always passes the flags *-X*, *-NL*, and *-L/usr/lib* to the loader. The compiler flags *-fi* and *-fn* are passed to the loader (in addition to the effects they have on the compiler). The compiler option *-o* causes a similar *-o* option to be passed to *ld*. *-Eposix* is passed to the loader except when *-pcc* is given, in which case *-Enoposix* is passed.

The compiler compatibility mode controls the libraries searched by the loader; this is normally accomplished by passing one or more *-l* options to the loader.

The following options, when present on the *cc* command line are passed to *ld*(1):

-A -D -E -F -L -M -T -X -d -e -l -m -r -s -t -u -x -y

See the *CONVEX LOADER User's Guide* for their meaning and use. The *-l* option must be specified after all object files on the *cc* command line to be effective. Loader options which require a value (e.g., *-L* and *-E*) must be written with no spaces between the flag and value (e.,g., -L/mydir) when passed through *cc*.

The *-link* option causes the following argument to be passed to *ld*. If the following argument does not start with - or starts with *-l* the argument is added to the loader's file list, otherwise it is added to the loader's flag list. For example, *-link -v3.2.8.5 -link -link* passes the flag *-v3.2.8.5* to the loader causing it to set the version number of the executable, and passes *-link* in the file list causing the loader to search the library libink.a.

OBJECT FILE COMPATIBILITY

When invoked without the *-pcc* flag CONVEX C V4.0 generates object files which are not compatible with object files created by previous compilers.

When invoked with the *-pcc* flag, object files created by previous C compilers may be linked with those created by V4.0; the *-pcc* flag must be used when linking the objects and the cc V4.0 compiler must be used to perform the link step.

Object files created by any mode of CONVEX C V4.0 may be mixed with object files created by any other mode of CONVEX C V4.0.

Object files created by CONVEX C V4.0 should not be linked using other C compilers.

ENVIRONMENT VARIABLES

The CONVEX C compiler prepends the value of the environment variable **CCOPTIONS** (if it is set) to each command line so that options need not be specified every time *cc* is invoked. For example,

```
setenv CCOPTIONS '-O2'
```

causes all compilations to be done at *-O2* and

```
setenv CCOPTIONS '-pcc'
```

causes all compilations to be done in backwards compatible mode.

The preprocessor *cpp* has a similar variable *CPPOPTIONS* which can be used to affect its behavior when it is invoked directly. However, *CPPOPTIONS* does not have any effect when the compiler invokes *cpp* (the compiler removes the variable before invoking *cpp*).

FILES

file.c	C language input file
file.o	object code output file
a.out	executable output file
/tmp/cocc*	temporary file
/lib/cpp	preprocessor
/usr/lib/cc/cocc	compiler
/usr/lib/cc/errmsgc	compiler error message text
/bin/cc	compiler control program
/usr/lib/crt/crt0.o	runtime start off
/usr/lib/crt/mcrt0.o	startup routine for prof-profiling
/usr/lib/crt/gcrt0.o	startup routine for gprof-profiling
/usr/lib/crt/bcrt0.o	startup routine for bprof-profiling
/usr/lib/libc_old.a	backward mode library
/usr/lib/libc.a	extended mode library
/usr/lib/libp1.a	standard mode library
/usr/lib/libansic.a	strictly conforming mode library
/usr/lib/libC1.a	C1 machine dependent library
/usr/lib/libC2.a	C2 machine dependent library
/usr/lib/libC1_old.a	backward compatible C1 machine dependent library
/usr/lib/libC2_old.a	backward compatible C2 machine dependent library
/usr/include	standard directory for "#include" files
/lib/bscan	optional bprof scanner
mon.out	file produced for analysis by *prof*(1)
gmon.out	file produced for analysis by *gprof*(1)
bmon.out	file produced for analysis by *bprof*(1)

Each library has a profiled version whose name is formed by by inserting _p before the .a.

BUGS

See the CONVEX C release notes in /usr/doc for a description of known bugs causing wrong answers.

The compiler will generate incorrect code for a program containing a function call such as f(g(),g()) if g() returns a struct.

A program containing code such as { int a; { int b = &a; int a;}} will have *b* incorrectly initialized to the address of the innermost *a*.

SEE ALSO

adb(1), as(1), cpp(1), csd(1)(optional product), gprof(1)(optional product), ld(1), prof(1)(optional product), bprof(1)(optional product), monitor(3), a.out(5)

CONVEX ANSI C Concepts
CONVEX C User's Guide
CONVEX C Language Reference Manual
CONVEX C Optimization Guide
CONVEX C Quick Reference
CONVEX Loader User's Guide
B. W. Kernighan and D. M. Ritchie, *The C Programming Language, Second Edition,* Prentice-Hall, 1988
B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, 1978
B. W. Kernighan, *Programming in C—a tutorial*

DIAGNOSTICS

The *CONVEX C User's Guide* provides detailed information regarding error messages produced by the compiler.

A successful compilation is indicated by returning status 0.

Occasional messages may be produced by the assembler or loader.

NAME

cpp – C preprocessor

SYNOPSIS

/usr/lib/cc/cpp [*option*] ... [*infile* [*outfile*]]

DESCRIPTION

cpp is the macro preprocessor used by the CONVEX C compiler *cc*(1). *cpp* is an ANSI C compatible preprocessor that can also be made compatible with previous versions of *cpp* by using the *-pcc* option (described below).

COMMAND LINE OPTIONS

If no files are specified *cpp* reads lines from *stdin*, processes them, and writes them to *stdout*. *infile* and *outfile* may be specified to change this. There is no way to specify an output file without specifying an input file.

The following command line options are interpreted by *cpp*:

–C Retain comments in the output. C style comments are usually removed.

–D*name*
–D*name*=*def*
Define the *name* (as if by *#define*) as *def*, or 1 if *def* is omitted.

–I*dir* Add *dir* to the search list for *#include* files.

–P Don't insert control lines into the output.

–R Allow recursive macros.

–U*name* Remove any initial definition of *name*. Names predefined by the preprocessor are discussed below.

–pcc Behave compatibly with earlier preprocessors that were not ANSI C conforming.

–vn Identifies version. Outputs the version number of *cpp* to *stderr*.

Lines in the input file that have a "#" character as the first nonblank character are interpreted as directives to *cpp*. The effect of a directive lasts until the end of the source or until it is countered by another directive.

ENVIRONMENT VARIABLES

The CONVEX C preprocessor *cpp* prepends the value of the environment variable **CPPOPTIONS** (if it is set) to each command line so that options need not be specified every time *cpp* is invoked. For example,

```
setenv CPPOPTIONS '-pcc'
```

causes all preprocessing to occur in backwards compatible mode.

The CONVEX C compiler removes the value of *CPPOPTIONS* before invoking the preprocessor to avoid conflict. *CPPOPTIONS* is useful when invoking *cpp* as a general purpose preprocessor.

PREDEFINED NAMES

The names that are predefined by the preprocessor depend on the options used.

The identifiers "__convex__" and "__unix__" are always predefined as 1.

The identifiers "__LINE__" and "__FILE__" are always predefined; they return the current line number and file name respectively. These definitions may not be removed with #undef or **-U**.

In ANSI conforming mode the identifiers "__DATE__" and "__TIME__" are predefined; they return the current date and time respectively. These definitions may not be removed with #undef or **-U**.

When invoked with the *-pcc* flag the names "convex" and "unix" are predefined. Applications should use __convex__ and __unix__ instead of these names.

When invoked by the CONVEX C compiler other names may be predefined. See *cc(1)* for details.

Other names beginning with "__" may be predefined by *cpp*. Such names are reserved to CONVEX; their usage or availability may change in subsequent releases. Applications should not depend on the *presence or absence* of names beginning with "__" (except those defined above).

TOKEN REPLACEMENT

A control line of the form:

#define *identifier replacement-string*

is a simple macro definition. It causes *cpp* to replace subsequent instances of the identifier with the replacement-string. A line of the form:

#define *identifier(argument, ...) replacement-string*

is a macro definition with arguments. Subsequent instances of the identifier are replaced by the replacement-string in the definition. Each occurrence of an argument mentioned in the formal parameter list of the definition is replaced by the corresponding actual argument from the call.

The ## operator, which may appear only in the replacement-string of #define directive, causes catenation of its operands. Thus

```
#define x a ## b
```

is equivalent to

```
#define x ab
```

This operator is not available when *-pcc* is specified.

The # operator may only appear immediately before a formal parameter name in the replacement-string of a #define directive. When expansion of the macro occurs the actual parameter becomes a string constant. Thus

```
#define f(x)    #x[0]
f(hello)
```

produces

```
"hello"[0]
```

This operator is not available when *-pcc* is specified.

A long macro definition may be continued on another line by adding "\" at the end of the line to be continued.

The replacement string is rescanned for more defined identifiers.

A control line of the form:

#undef *identifier*

causes the identifier's macro definition to be deleted.

FILE INCLUSION

A control line of the form:

#include "*filename*"

or

#include <*filename*>

causes the replacement of that line by the entire contents of *filename*. Files included by a #include statement may contain further *#include* statements; include files may be nested.

The directory search order for *#include* files is (1) the directory of the file that contains the request unless the form **#include** <*filename*> was used (2) the directories specified by *-I*, and (3) the standard directory (*/usr/include*).

CONDITIONAL INSERTION

A control line of the form:

#if *constant-expression*

checks whether the constant expression evaluates to a nonzero quantity. A constant expression is any legitimate C expression that evaluates to a constant. See *"The C Programming Language"* for more information. A control line of the form:

#ifdef *identifier*

checks whether the identifier is currently defined in *cpp*. A control line of the form:

#ifndef *identifier*

checks whether the identifier is currently undefined in *cpp*.

Each form of the if statement may be followed by an arbitrary number of lines of text, optionally followed by

#elif *constant-expression*

and more lines of text. Multiple *#elif* constructs may be used.

After all the #elif lines a

#else

line may appear. The conditional input section is terminated by a

#endif

line.

The text is included or omitted according to the truth of the expression.

The expression on a #if or #elif line is subject to macro expansion.

These constructs may be nested.

LINE CONTROL

For other preprocessors that generate C programs, a line of the form:

#line *integer-constant string-constant*

causes the C compiler to believe, for purposes of error diagnostics, that the number of the next source line is given by the integer-constant and the current input file is given by the string-constant. If the string-constant is absent, the current filename does not change.

COMMENTS

C comments are replaced by a blank unless the *-C* option or the *-pcc* option is specified.

When the -C option is specified comments are retained in the output.

When the *-pcc* option is specified (without -C) comments are removed from the text (no blank is introduced in place of the comment). This feature is often used to catenate two pieces of text; such usage is not portable. Thus, in " `foo/* comment */bar` ", the output will be "foobar" unless either "foo" or "bar" is a macro name. Expansion of "foo" and "bar" will take place if they are macro names. No macro expansion of "foobar" will be performed.

OTHER CONSIDERATIONS

The following features are not portable and are supported only with the -pcc flag.

Formal macro parameters are recognized in *# define* token strings, even inside character constants and quoted strings. The output from:

```
#define foo(a) '\a'
foo(bar)
```

is the six characters `'\bar'`.

Macro names are not recognized inside character constants or quoted strings during the regular scan. Thus:

```
#define foo bar
printf("foo");
```

does not expand "foo" in the second line, because it is inside a quoted string that is not part of a *# define* or *# undef.*

A mismatch between the number of formals and actuals in a macro call produces only a warning, not an error. Excess actuals are ignored; missing actuals are turned into null strings.

SEE ALSO

cc(1)

Draft Proposed ANSI Programming Language C (X3J11/88-159)

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*

B. W. Kernighan and D. M. Ritchie, *The C Programming Language, Second Edition*

NAME

vc – CONVEX C compiler

DESCRIPTION

vc is the name used to invoke versions of the CONVEX C compiler with version numbers of 3.0 and less. Version 4.0 of the CONVEX C compiler is invoked as *cc*. See the **cc** man page for more information.

NAME

vcpp – CONVEX C preprocessor

DESCRIPTION

vcpp is the name used to invoke versions of the CONVEX C preprocessor with version numbers of 3.0 and less. Version 4.0 of the CONVEX C preprocessor is invoked as *cpp*. See the **cpp** man page for more information.

NAME

intro – introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible return value. This is almost always **–1**; the individual descriptions specify the details.

As with normal arguments, all return codes and values from functions are of type integer unless otherwise noted. An error number is also made available in the external variable *errno*, which is not cleared on successful calls. Thus, *errno* should be tested only after an error has occurred.

The subroutine *perror*(3) produces a short error message on the standard error output describing the type of error that has occurred. We strongly recommend comprehensive usage of this routine upon error detection in system call returns.

The following is a complete list of the errors and their names as given in *<errno.h>*.

0 Error 0
Unused.

1 EPERM Not owner
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or superuser. It is also returned for attempts by ordinary users to do things allowed only to the superuser.

2 ENOENT No such file or directory
This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a pathname does not exist.

3 ESRCH No such process
The process whose number was given to *kill*(2) or *pattach*(2) does not exist, or is already dead.

4 EINTR Interrupted system call
An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

5 EIO I/O error
Some physical I/O error occurred during a *read* or *write*. This error may, in some cases, occur on a call following the one to which it actually applies.

6 ENXIO No such device or address
I/O on a special file refers to a subdevice which does not exist, or is beyond the limits of the device. It may also occur when, for example, an illegal tape drive unit number is selected or a disk pack is not loaded on a drive.

7 E2BIG Arg list too long
An argument list longer than 10240 bytes is presented to *execve*.

8 ENOEXEC Exec format error
A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number (see *a.out*(5)).

9 EBADF Bad file number
Either a file descriptor refers to no open file, or a read (resp. write) request is made to a file which is open only for writing (resp. reading).

10 ECHILD No children
wait and the process has no living or unwaited-for children.

11 EAGAIN No more processes
In a *fork,* the system's process table is full or the user is not allowed to create any more processes.

12 ENOMEM Not enough memory
During an *execve* or *break,* a program asks for more core or swap space than the system is able to supply. A lack of swap space is normally a temporary condition, however a lack of core is not a temporary condition; the maximum size of the text, data, and stack segments is a system parameter.

13 EACCES Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

14 EFAULT Bad address
The system encountered an invalid address in attempting to access the arguments of a system call.

15 ENOTBLK Block device required
A plain file was mentioned where a block device was required (e.g., in *mount*).

16 EBUSY Mount device busy
An attempt to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file directory (open file, current directory, mounted-on file, or active text segment).

17 EEXIST File exists
An existing file was mentioned in an inappropriate context (e.g., *link*).

18 EXDEV Cross-device link
A hard link to a file on another device was attempted.

19 ENODEV No such device
An attempt was made to apply an inappropriate system call to a device (e.g., read a write-only device).

20 ENOTDIR Not a directory
A non-directory was specified where a directory is required (e.g., in a pathname or as an argument to *chdir*).

21 EISDIR Is a directory
An attempt to write on a directory.

22 EINVAL Invalid argument
Some invalid argument (e.g., dismounting a non-mounted device, mentioning an unknown signal in *signal,* reading or writing a file for which *seek* has generated a negative pointer, etc.). Also set by math functions (see *intro*(3)).

23 ENFILE File table overflow
The system's table of open files is full, and temporarily no more *opens* can be accepted.

24 EMFILE Too many open files
Customary configuration limit is 20 per process.

25 ENOTTY Not a typewriter
The file mentioned in an *ioctl* is not a terminal or one of the other devices to which these calls apply.

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program which is currently open for writing or reading. Also an attempt to open a pure-procedure program that is being executed.

27 EFBIG File too large
The size of a file exceeded the maximum (about 10^9 bytes).

28 ENOSPC No space left on device
During a *write* to an ordinary file, there is no free space left on the device.

29 ESPIPE Illegal seek
An *lseek* was issued to a pipe. This error may also be issued for other non-seekable devices.

30 EROFS Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK Too many links
An attempt to make more than 32767 hard links to a file.

32 EPIPE Broken pipe
A write on a pipe or socket for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

33 EDOM Math argument
The argument of a function in the math package (3M) is out of the domain of the function.

34 ERANGE Result too large
The value of a function in the math package (3M) is unrepresentable within machine precision.

35 EWOULDBLOCK Operation would block
An operation which would cause a process to block was attempted on a object in non-blocking mode (see *ioctl*(2)).

36 EINPROGRESS Operation now in progress
An operation which takes a long time to complete (such as a *connect*(2)) was attempted on a non-blocking object (see *ioctl*(2)).

37 EALREADY Operation already in progress
An operation was attempted on a non-blocking object which already had an operation in progress.

38 ENOTSOCK Socket operation on non-socket
Self-explanatory.

39 EDESTADDRREQ Destination address required
A required address was omitted from an operation on a socket.

40 EMSGSIZE Message too long
A message sent on a socket was larger than the internal message buffer.

41 EPROTOTYPE Protocol wrong type for socket
A protocol was specified which does not support the semantics of the socket type requested. For example you cannot use the ARPA Internet UDP protocol with type SOCK_STREAM.

42 ENOPROTOOPT Bad protocol option
A bad option was specified in a *getsockopt*(2) or *setsockopt*(2) call.

43 EPROTONOSUPPORT Protocol not supported
The protocol has not been configured into the system, or no implementation for it exists.

44 ESOCKTNOSUPPORT Socket type not supported
The support for the socket type has not been configured into the system, or no implementation for it exists.

45 EOPNOTSUPP Operation not supported on socket
For example, trying to *accept* a connection on a datagram socket.

46 EPFNOSUPPORT Protocol family not supported
The protocol family has not been configured into the system, or no implementation for it exists.

47 EAFNOSUPPORT Address family not supported by protocol family
An address incompatible with the requested protocol was used. For example, you should not necessarily expect to be able to use PUP Internet addresses with ARPA Internet protocols.

48 EADDRINUSE Address already in use
Only one usage of each address is normally permitted.

49 EADDRNOTAVAIL Can't assign requested address
Normally results from an attempt to create a socket with an address not on this machine.

50 ENETDOWN Network is down
A socket operation encountered a dead network.

51 ENETUNREACH Network is unreachable
A socket operation was attempted to an unreachable network.

52 ENETRESET Network dropped connection on reset
The host you were connected to crashed and rebooted.

53 ECONNABORTED Software caused connection abort
A connection abort was caused internal to your host machine.

54 ECONNRESET Connection reset by peer
A connection was forcibly closed by a peer. This normally results from the peer executing a *shutdown*(2) call.

55 ENOBUFS No buffer space available
An operation on a socket or pipe was not performed because the system lacked sufficient buffer space.

56 EISCONN Socket is already connected
A *connect* request was made on an already connected socket; or, a *sendto* or *sendmsg* request on a connected socket specified a destination other than the connected party.

57 ENOTCONN Socket is not connected
A request to send or receive data was disallowed because the socket was not connected.

58 ESHUTDOWN Can't send after socket shutdown
A request to send data was disallowed because the socket had already been shut down with a previous *shutdown*(2) call.

59 ETOOMANYREFS Too many references: can't splice

60 ETIMEDOUT Connection timed out
A *connect* request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)

61 ECONNREFUSED Connection refused
No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service which is inactive on the foreign host.

62 ELOOP Too many levels of symbolic links
A pathname lookup involved more than 8 symbolic links.

63 ENAMETOOLONG File name too long
A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.

64 EHOSTDOWN Host is down

65 EHOSTUNREACH No route to host

66 ENOTEMPTY Directory not empty
A directory with entries other than . and .. was supplied to a remove directory or rename call.

67 EPROCLIM Too many processes

68 EUSERS Too many users

69 EDQUOT Disc quota exceeded

70 ESTALE Stale NFS file handle
A client referenced an open file, when the file has been deleted.

71 EREMOTE Too many levels of remote in path
An unsupported operation was attempted on a file within an NFS filesystem, such as attempting to get the file's "file handle" (see *getfh*(2)).

72 EDEADLK Deadlock situation detected/avoided
The *lockf*(3) library frontend routine returns this if the *fcntl*(2) system call returns the ENOLCK error for compatibility with */usr/group* standards.

73 ENOLCK No record locks available
The record lock daemon process *lockd*(8C) has not been started or all record lock resources have been consumed.

74 ENOSYS Function not implemented

75 ETPTRUNC Logical tape record would be truncated
A particular *write*(2) to a labeled tape would result in a logical record being truncated because its length is greater than the maximum record size.

76 ETPBADFORM Labeled tape not in proper format
A label or tapemark was not found where it was expected. Continuing reading after receiving this error may give unpredictable results.

77 ETPNOSUPPORT Tape feature not supported
A perhaps valid tape feature (such as a binary-coded variable record format (V)) was discovered which is not supported on this system.

DEFINITIONS

Process ID
Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 0 to 30000.

Parent process ID
A new process is created by a currently active process (see *fork*(2)). The parent process ID of a process is the process ID of its creator.

Process Group ID
Each active process is a member of a process group that is identified by a positive integer called the process group ID. This is the process ID of the group leader. This grouping permits the signalling of related processes (see *killpg*(2)) and the job control mechanisms of *csh*(1).

Sessions
Each process group is a member of a session; a process is considered a member of the session

of which its process group is a member. A new process (see *fork*(2)) joins the session of its parent. A process can use *setsid*(2)) to alter its session membership; it thus becomes a session leader, and as such, may cause a terminal to become its controlling terminal (see *tty*(4)).

Tty Group ID

Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to arbitrate between multiple jobs contending for the same terminal (see *csh*(1) and *tty*(4)).

Real User ID and Real Group ID

Each user on the system is identified by a positive integer termed the real user ID.

Each user is also a member of one or more groups. One of these groups is distinguished from others and used in implementing accounting facilities. The positive integer corresponding to this distinguished group is termed the real group ID.

All processes have a real user ID and a real group ID. These are initialized from the equivalent attributes of the process which created it.

Effective User ID, Effective Group ID, and Access Groups

Access to system resources is governed by three values: the effective user ID, the effective group ID, and the group access list.

The effective user ID and effective group ID are initially the process' real user ID and real group ID, respectively. Either may be modified through execution of a set-user-ID or set-group-ID file, possibly by one of its ancestors (see *execve*(2)).

The group access list is an additional set of group ID's used only in determining resource accessibility. Access checks are performed as described below in **File Access Permissions**

Activity ID

Each process has an activity ID associated with it. Upon login, activity ID is initialized to 0; subsequently, each process inherits the activity ID of its parent. The activity ID provides a means by which system managers may design customized billing categories for use in conjunction with groups. Various user programs record activity ID in their log files, resulting in a global, yet flexible, resource accounting system. The activity ID is set using the system call *setaid*(2), which is available only to the superuser. (See *bill*(1) for information about how a user may set his activity ID without being the superuser; see also *activities*(5)).

Superuser

A process is recognized as a *superuser* process and is granted special privileges if its effective user ID is 0.

Special Processes

The processes with a process ID's of 0, 1, and 2 are special. Process 0 is the scheduler. Process 1 is the initialization process *init*, and is the ancestor of every other process in the system. It is used to control the process structure. Process 2 is the paging daemon.

Descriptor

An integer assigned by the system when a file is referenced by *open*(2), *dup*(2), or *pipe*(2) or a socket is referenced by *socket*(2) or *socketpair*(2), which uniquely identifies an access path to that file or socket from a given process or any of its children.

Filename

Names consisting of up to {FILENAME_MAX} characters may be used to name an ordinary file, special file, or directory.

These characters may be selected from the set of all ASCII character excluding **0** (null) and the ASCII code for /(slash). (The parity bit, bit 8, must be 0.)

Note that it is generally unwise to use *, !, ?, [, or] as part of filenames because of the special meaning attached to these characters by the shell.

Path Name

A path name is a null-terminated character string starting with an optional slash /, followed by zero or more directory names separated by slashes, optionally followed by a file name. The total length of a path name must be less than {PATHNAME_MAX} characters.

If a path name begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory. A slash by itself names the root directory. A null pathname refers to the current directory.

Directory

A directory is a special type of file that contains entries that are references to other files. Directory entries are called links. By convention, a directory contains at least two links, . and .., referred to as *dot* and *dot-dot*, respectively. *Dot* refers to the directory itself and *dot-dot* refers to its parent directory.

Root Directory and Current Working Directory

Each process has associated with it, a concept of a root directory and a current working directory for the purpose of resolving path name searches. A process' root directory need not be the root directory of the root file system.

File Access Permissions

Every file in the file system has a set of access permissions. These permissions are used in determining whether a process may perform a requested operation on the file (such as opening a file for writing). Access permissions are established when a file is created. They may be changed at some later time through the *chmod*(2) call.

File access is broken down according to whether a file may be: read, written, or executed. Directory files use the execute permission to control if the directory may be searched.

File access permissions are interpreted by the system as they apply to three different classes of users: the owner of the file, users in the file's group, and anyone else. Every file has an independent set of access permissions for each of these classes. When an access check is made, the system decides if permission should be granted by checking the access information applicable to the caller.

Read, write, and execute/search permissions on a file are granted to a process if the process' effective user ID is that of the superuser. For non-superusers, additional access checks are made. If the process' effective user ID matches that of the owner of the file, permission is denied unless the owner permissions allow the access. If the process' effective group ID matches the group ID of the file, permission is denied unless the group permissions allow the access. If the group ID of any group to which the process belongs (see *getgroups(2)*) matches the group ID of the file, permission is denied unless the group permissions allow the access. If neither the user ID nor any of the group IDs matches, permission is denied unless the permissions for "other users" allow access.

Sockets and Address Families

A socket is an endpoint for communication between processes. Each socket has queues for sending and receiving data.

Sockets are typed according to their communications properties. These properties include whether messages sent and received at a socket require the name of the partner, whether communication is reliable, the format used in naming message recipients, etc.

Each instance of the system supports some collection of socket types (see *socket*(2) for more information about the types available and their properties).

Each instance of the system supports some number of sets of communications protocols. Each protocol set supports addresses of a certain format. An Address Family is the set of

addresses for a specific group of protocols. Each socket has an address chosen from the address family in which the socket was created.

SEE ALSO

intro(3), perror(3)

NAME

accept – accept a connection on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

ns = accept(s, addr, addrlen)
int ns, s;
struct sockaddr *addr;
int *addrlen;
```

DESCRIPTION

The argument *s* is a socket that has been created with *socket*(2), bound to an address with *bind*(2), and is listening for connections after a *listen*(2). *Accept* extracts the first connection on the queue of pending connections, creates a new socket with the same properties of *s* and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, *accept* blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, *accept* returns an error as described below. The accepted socket, *ns*, may not be used to accept more connections. The original socket *s* remains open.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

It is possible to *select*(2) a socket for the purposes of doing an *accept* by selecting it for read.

RETURN VALUE

The call returns –1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

ERRORS

The *accept* will fail if:

[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The descriptor references a file, not a socket.
[EOPNOTSUPP]	The referenced socket is not of type SOCK_STREAM.
[EFAULT]	The *addr* parameter is not in a writable part of the user address space.
[EWOULDBLOCK]	The socket is marked non-blocking and no connections are present to be accepted.

SEE ALSO

bind(2), connect(2), listen(2), select(2), socket(2)

NAME

access – determine accessibility of file

SYNOPSIS

```
#include <unistd.h>

int access(path, mode)
int accessible;
char *path;
int  mode;
```

DESCRIPTION

access() checks the given file *path* for accessibility according to *mode*, which is an inclusive or of the bits R_OK, W_OK and X_OK. Specifying *mode* as F_OK (i.e. 0) tests whether the directories leading to the file can be searched and the file exists.

The real user ID and the group access list (including the real group ID) are used in verifying permission, so this call is useful to set-UID programs.

Notice that only access bits are checked. A directory may be indicated as writable by *access*, but an attempt to open it for writing will fail (although files may be created there); a file may look executable, but *execve* will fail unless it is in proper format.

RETURN VALUE

If *path* cannot be found or if any of the desired access modes would not be granted, then a -1 value is returned; otherwise a 0 value is returned.

ERRORS

Access to the file is denied if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	The argument path name was too long.
[ENOENT]	The *path* argument points to an empty string.
[ENOENT]	Read, write, or execute (search) permission is requested for a null path name or the named file does not exist.
[EINVAL]	The *path* argument contains a byte with the high-order bit set.
[EINVAL]	The *mode* argument contains a bit other than a combination of F_OK, R_OK, W_OK, and X_OK.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EROFS]	Write access is requested for a file on a read-only file system.
[ETXTBSY]	Write access is requested for a pure procedure (shared text) file that is being executed.
[EACCES]	Permission bits of the file mode do not permit the requested access; or search permission is denied on a component of the path prefix. The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits, members of the file's group other than the owner have permission checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.
[EFAULT]	*path* points outside the process's allocated address space.

NOTES

Remote accesses to local files using the *access* system call cannot be detected by the *faillogon* utility.

BACKWARD COMPATIBILITY

Previous versions of the operating system accepted an empty *path* string as a synonym for the current directory.

SEE ALSO

chmod(2), stat(2), faillogon(8)

NAME

acct – turn accounting on or off

SYNOPSIS

acct(file)
char *file;

DESCRIPTION

The system is prepared to write a record in an accounting *file* for each process as it terminates. This call, with a null-terminated string naming an existing file as argument, turns on accounting; records for each terminating process are appended to *file*. An argument of 0 causes accounting to be turned off.

The accounting file format is given in *acct*(5).

If accounting is already on when accounting is turned on, the accounting log will be atomically switched to the new file.

This call is permitted only to the super-user.

NOTES

Accounting is automatically disabled when the file system the accounting file resides on runs out of space; it is enabled when space once again becomes available.

RETURN VALUE

On error –1 is returned. The file must exist and the call may be exercised only by the super-user.

ERRORS

Acct will fail if one of the following is true:

[EPERM]	The caller is not the super-user.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EISDIR]	The named file is a directory.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*File* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EACCES]	The file is a character or block special file.

SEE ALSO

acct(5), sa(8)

BUGS

No accounting is produced for programs running when a crash occurs. In particular nonterminating programs are never accounted for.

NAME

adjtime – adjust time

SYNOPSIS

#include <sys/time.h>

adjtime(tp, otp)
struct timeval *tp, *otp;

DESCRIPTION

adjtime adjusts the system's notion of the current time. The time is adjusted by the amount of time in *tp. The old adjustment value is returned in *otp.

The adjustment is effected by speeding up or slowing down the system's clock by a fixed percentage, currently 10%.

The structures pointed to by *tp* and *otp* are defined in *<sys/time.h>* as:

```
struct timeval {
        u_long  tv_sec;         /* seconds since Jan. 1, 1970 */
        long    tv_usec;        /* and microseconds */
};
```

If *otp* is a zero pointer, the corresponding information will not be returned.

Only the superuser may adjust the time of day.

The adjustment value will be silently rounded to the resolution of the system clock.

RETURN

A 0 return value indicates that the call succeeded. A –1 return value indicates an error occurred, and in this case an error code is stored into the global variable *errno*.

ERRORS

The following error codes may be set in *errno*:

[EFAULT] An argument address referenced invalid memory.

[EPERM] A user other than the superuser attempted to set the time.

SEE ALSO

settimeofday(2), date(1)

NAME

asiostat – wait, then return asynchronous I/O byte count

SYNOPSIS

retval = asiostat(*fd*)
int retval, *fd*;

DESCRIPTION

Asiostat first waits until any outstanding asynchronous I/O on file descriptor *fd* has been completed, and then returns the sum of the number of bytes asynchronously transferred to or from the file by the requesting process since the last call to *asiostat*.

RETURN VALUE

If errors were detected during any of the previous asynchronous I/O operations since the last *asiostat* call, a -1 is returned and the global variable *errno* will contain the first error recorded. If no errors were detected, the sum of the number of bytes asynchronously transferred since the last *asiostat* is returned.

ERRORS

For the interpretation of errors, see *read* or *write* as appropriate. *Asiostat* will itself return EBADF for nonvalid file descriptors, or EFAULT if *astat* points outside of the allocated address space.

SEE ALSO

fcntl(2), read(2), write(2)

NAME

bind – bind a name to a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

bind(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

DESCRIPTION

Bind assigns a name to an unnamed socket. When a socket is created with *socket*(2) it exists in a name space (address family) but has no name assigned. *Bind* requests that *name* be assigned to the socket.

NOTES

Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using *unlink*(2)).

The rules used in name binding vary between communication domains. Consult the manual entries in section 4 for detailed information.

RETURN VALUE

If the bind is successful, a 0 value is returned. A return value of –1 indicates an error, which is further specified in the global *errno*.

ERRORS

The *bind* call will fail if:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is not a socket.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address.
[EACCES]	The requested address is protected, and the current user has inadequate permission to access it.
[EFAULT]	The *name* parameter is not in a valid part of the user address space.

The following errors are specific to binding names in the UNIX domain.

[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	A prefix component of the path name does not exist.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EROFS]	The name would reside on a read-only file system.
[EISDIR]	A null pathname was specified.

SEE ALSO

connect(2), listen(2), socket(2), getsockname(2)

NAME

brk, sbrk – change data segment size

SYNOPSIS

caddr_t brk(addr)
caddr_t addr;

caddr_t sbrk(incr)
int incr;

DESCRIPTION

Brk sets the system's idea of the lowest data segment location not used by the program (called the break) to *addr* (rounded up to the next multiple of the system's page size). Locations greater than *addr* rounded up to the next multiple of the system's page size and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

In the alternate function *sbrk*, *incr* more bytes are added to the program's data space and a pointer to the start of the new area is returned. The returned pointer is not page aligned.

When a program begins execution via *execve* the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use *sbrk*.

The *getrlimit*(2) system call may be used to determine the maximum permissible size of the *data* segment; it will not be possible to set the break beyond the *rlim_max* value returned from a call to *getrlimit*, e.g. "etext + rlp→rlim_max." (See *end*(3) for the definition of *etext*.)

RETURN VALUE

Zero is returned if the *brk* could be set; –1 if the program requests more memory than the system limit. *Sbrk* returns –1 if the break could not be set.

ERRORS

Sbrk will fail and no additional memory will be allocated if one of the following are true:

[ENOMEM] The limit, as set by *setrlimit*(2), was exceeded.

[ENOMEM] The maximum possible size of a data segment (compiled into the system) was exceeded.

[ENOMEM] Insufficient space existed in the swap area to support the expansion.

SEE ALSO

execve(2), getrlimit(2), malloc(3), end(3)

BUGS

Setting the break may fail due to a temporary lack of swap space. It is not possible to distinguish this from a failure caused by exceeding the maximum size of the data segment without consulting *getrlimit*.

NAME

chdir – change current working directory

SYNOPSIS

chdir(path)
char *path;

DESCRIPTION

path is the pathname of a directory. *chdir()* causes this directory to become the current working directory, the starting point for path names not beginning with "/."

In order for a directory to become the current directory, a process must have execute (search) access to the directory.

If *chdir()* fails, the current directory is unchanged.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

chdir() will fail and the current working directory will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the pathname is not a directory.
[ENOENT]	The named directory does not exist, or *path* is an empty string.
[ENAMETOOLONG]	The argument path name was too long.
[EINVAL]	The argument contains a byte with the high-order bit set.
[EACCES]	Search permission is denied for any component of the path name.
[EFAULT]	*Path* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

BACKWARD COMPATIBILITY

Previous versions of the operating system allowed an empty pathname string as a synonym for the current directory.

SEE ALSO

chroot(2)

NAME

chmod – change mode of file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
chmod(path, mode)
char *path;
mode_t mode;
```

DESCRIPTION

The file whose name is given by *path* has its mode changed to *mode*. Modes are constructed by *or*ing together some combination of the following:

S_ISUID	set effective user ID on execution
S_ISGID	set effective group ID on execution
S_IRWXU	read, write, execute (search) by owner
S_IRUSR	read by owner
S_IWUSR	write by owner
S_IXUSR	execute (search on directory) by owner
S_IRWXG	read, write, execute (search) by group
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute (search on directory) by group
S_IRWXO	read, write, execute (search) by others
S_IROTH	read by others
S_IWOTH	write by others
S_IXOTH	execute (search on directory) by others

Only the owner of a file (or the super-user) may change the mode.

Writing or changing the owner of a file turns off the set-user-ID and set-group-ID bits. This makes the system somewhat more secure by protecting set-user-ID (set-group-ID) files from remaining set-user-ID (set-group-ID) if they are modified, at the expense of a degree of compatibility.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

chmod() will fail and the file mode will be unchanged if:

[EINVAL]	The *path* argument contains a byte with the high-order bit set.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	The pathname was too long.
[ENOENT]	The named file does not exist, or the *path* argument points to an empty string.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not the super-user.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

CONVEX EXTENSIONS

Convex adds the following mode bit.

_S_ISVTX see *sticky*(8)

SEE ALSO

fchmod(2), open(2), chown(2)

NAME

chown – change owner and group of a file

SYNOPSIS

```
#include <sys/types.h>

chown(path, owner, group)
char *path;
uid_t owner;
gid_t group;
```

DESCRIPTION

The file that is named by *path* has its *owner* and *group* changed as specified. Only the super-user may change the owner of the file, because if users were able to give files away, they could defeat the file-space accounting procedures. The owner of the file may change the group to a group of which he is a member.

On some systems, *chown()* clears the set-user-ID and set-group-ID bits on the file to prevent accidental creation of set-user-ID and set-group-ID programs owned by the super-user.

RETURN VALUE

Zero is returned if the operation was successful; –1 is returned if an error occurs, with a more specific error code being placed in the global variable *errno*.

ERRORS

chown() will fail and the file will be unchanged if:

[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EPERM]	The effective user ID is not the super-user and either the file owner was being changed or the effective user ID does not match the file owner or the new group is not a member of the processes group list.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[EIO]	An I/O error occurred while reading from or writing to the file system.

CONVEX EXTENSIONS

A value of _SAME_UID passed for owner or _SAME_GID for group results in the current owner or group remaining unchanged.

If the final component of *path* is a symbolic link, the ownership and group of the symbolic link is changed, not the ownership and group of the file or directory to which it points.

BACKWARD COMPATIBILITY

The *owner* and *group* parameters were formerly declared to be type int, although the high word was unusable.

SEE ALSO

fchown(2), chown(8), chgrp(1), chmod(2), flock(2)

NAME

chroot – change root directory

SYNOPSIS

chroot(dirname)
char *dirname;

DESCRIPTION

Dirname is the address of the pathname of a directory, terminated by a null byte. *Chroot* causes this directory to become the root directory, the starting point for path names beginning with "/".

In order for a directory to become the root directory a process must have execute (search) access to the directory.

This call is restricted to the super-user.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate an error.

ERRORS

Chroot will fail and the root directory will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path name is not a directory.
[ENOENT]	The pathname was too long.
[EINVAL]	The argument contains a byte with the high-order bit set.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for any component of the path name.
[EFAULT]	*Path* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

SEE ALSO

chdir(2)

NAME

close – delete a descriptor

SYNOPSIS

int close(d)
int d;

DESCRIPTION

The *close()* call deletes a descriptor from the per-process object reference table. If this is the last reference to the underlying object, then it will be deactivated. For example, on the last close of a file the current *lseek()* pointer associated with the file is lost; on the last close of a *socket*(2) associated naming information and queued data are discarded; on the last close of a file holding an advisory lock the lock is released.

A close of all of a process's descriptors is automatic on *exit*, but since there is a limit on the number of active descriptors per process, *close* is necessary for programs which deal with many descriptors.

When a process *fork()*'s, all descriptors for the new child process reference the same objects as they did in the parent before the fork. If a new process is then to be run using *execve*(2), the process would normally inherit these descriptors. Most of the descriptors can be rearranged with *dup2*(2) or deleted with *close()* before the *execve()* is attempted, but if some of these descriptors will still be needed if the *execve()* fails, it is necessary to arrange for them to be closed if the *execve()* succeeds. For this reason, the call "fcntl(d, F_SETFD, FD_CLOEXEC)" is provided which arranges that a descriptor will be closed after a successful execve; the call "fcntl(d, F_SETFD, 0)" restores the default, which is to not close the descriptor.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and the global integer variable *errno* is set to indicate the error.

ERRORS

close() will fail if:

[EBADF]	*d* is not an active descriptor.
[EINTR]	The close function was interrupted by a signal.

BACKWARD COMPATIBILITY

The EINTR return value was formerly difficult to observe (see *sigvec*(2)). It is now the default, and the use of non-POSIX extensions is required to avoid it (see *sigaction*(2)).

SEE ALSO

accept(2), flock(2), open(2), pipe(2), socket(2), socketpair(2), execve(2), fcntl(2), fork(2)

NAME

connect – initiate a connection on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

connect(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

DESCRIPTION

The parameter *s* is a socket. If it is of type SOCK_DGRAM, then this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type SOCK_STREAM, then this call attempts to make a connection to another socket. The other socket is specified by *name*, which is an address in the communications space of the socket. Each communications space interprets the *name* parameter in its own way. Generally, stream sockets may successfully *connect* only once; datagram sockets may use *connect* multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

RETURN VALUE

If the connection or binding succeeds, then 0 is returned. Otherwise a –1 is returned, and a more specific error code is stored in *errno*.

ERRORS

The call fails if:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is a descriptor for a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available on this machine.
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
[EISCONN]	The socket is already connected.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	The network isn't reachable from this host.
[EADDRINUSE]	The address is already in use.
[EFAULT]	The *name* parameter specifies an area outside the process address space.
[EINPROGRESS]	The socket is non-blocking and the connection cannot be completed immediately. It is possible to *select*(2) for completion by selecting the socket for writing.
[EALREADY]	The socket is non-blocking and a previous connection attempt has not yet been completed.

The following errors are specific to connecting names in the UNIX domain. These errors may not apply in future versions of the UNIX IPC domain.

[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name

exceeded 1023 characters.

[ENOENT] The named socket does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] Write access to the named socket is denied.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENETDOWN] A network on the path to the local host is down.

[EHOSTDOWN] The remote host or a gateway to the remote host is down.

[EHOSTUNREACH]
The remote host is not reachable from this host.

SEE ALSO

accept(2), select(2), socket(2), getsockname(2)

NAME

creat – create a new file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
creat(name, mode)
char *name;
mode_t mode;
```

DESCRIPTION

creat() creates a new file or prepares to rewrite an existing file called *name*, given as the address of a null-terminated string. If the file did not exist, it is given mode *mode*, as modified by the process's mode mask. See *chmod*(2) for the construction of the *mode* argument.

If the file did exist, its mode and owner remain unchanged but it is truncated to 0 length.

The file is also opened for writing, and its file descriptor is returned.

RETURN VALUE

The value –1 is returned if an error occurs. Otherwise, the call returns a non-negative descriptor that only permits writing.

ERRORS

creat() will fail and the file will not be created or truncated if one of the following occur:

[EACCES]	A needed directory does not have search permission.
[EACCES]	The file does not exist and the directory in which it is to be created is not writable.
[EACCES]	The file exists, but it is unwritable.
[EEXIST]	O_CREAT and O_EXCL are set, and the file exists.
[EINTR]	The *creat()* operation was interrupted by a signal.
[EISDIR]	The file is a directory.
[EINVAL]	The *path* argument contains a byte with the high-order bit set.
[EMFILE]	There are already too many files open.
[ENAMETOOLONG]	The length of the *path* string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX while _POSIX_NO_TRUNC is in effect.
[ENFILE]	Too many files are currently open on the system.
[ENOENT]	A directory specified in the pathname does not exist.
[ENOENT]	The *path* argument points to the empty string.
[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	The named file resides on a read-only file system.
[ENXIO]	The file is a character special or block special file, and the associated device does not exist.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EFAULT]	*name* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

[EOPNOTSUPP]
The file was a socket (not currently implemented).

BACKWARD COMPATIBILITY

In previous releases of the operating system, the EINTR errno would not occur by default (see *sigvec(2)*).

In previous releases of the operating system, the empty pathname was a synonym for the current directory.

NOTES

The *mode* given is arbitrary; it need not allow writing. This feature has been used in the past by programs to construct a simple exclusive locking mechanism. It is replaced by the O_EXCL open mode, or *flock*(2) facility.

creat() is equivalent to

open(*path*,**O_WRONLY|O_CREAT|O_TRUNC,***mode*);

SEE ALSO

open(2), write(2), close(2), chmod(2), umask(2)

NAME

cvxprusage – get information about parallel resource utilization

SYNOPSIS

#include <sys/time.h>
#include <sys/resource.h>

cvxprusage(prusage)
struct cvxprusage *prusage;

DESCRIPTION

Cvxprusage returns information describing the resources utilized by the current process, with an emphasis on parallel resource utilization. The buffer that *prusage* points to will be filled in with the following structure:

```
struct cvxprusage {
  struct timeval pru_stime; /* system time, in secs and usecs */
  struct timeval pru_utime; /* user time,   in secs and usecs */
    /* user PC samples */
  unsigned long long pru_utotal;/* cumulative parallelism counts */
  unsigned long long pru_usamples;/* number parallelism samples */
    /* system PC samples */
  unsigned long long pru_stotal;/* cumulative parallelism counts */
  unsigned long long pru_ssamples;/* number parallelism samples */
  unsigned long long pru_pvtime;/* process virtual time */
    /* room for expansion */
  unsigned long long pru_filler[11];/* reserved for growth */
};
```

The fields are interpreted as follows:

pru_utime the total amount of CPU time spent executing in user mode. This is identical to the value returned in the ru_exutime field of the structure returned by *getrusage(2).*

pru_stime the total amount of CPU time spent in the system executing on behalf of the process. This is identical to the value returned in the ru_stime field of the structure returned by *getrusage(2).*

pru_usamples the number of times the system sampled the number of threads running in this process and found its program counter in user space.

pru_utotal the cumulative number of threads the system found for all samples taken while the progam's program counter was in user space.

pru_ssamples the number of times the system sampled the number of threads running in this process and found its program counter in system space.

pru_stotal the cumulative number of threads the system found for all samples taken while the progam's program counter was in system space.

pru_pvtime the process virtual time. This is the total wall time the process spent executing in user mode.

The system samples the number of threads running in a process at a fixed rate, currently 100 times per second. An estimate of user level process parallelism can be made by dividing *pru_utotal* by *pru_usamples.* User level parallelism within particular regions can be estimated by using the deltas of *pru_utotal* and *pru_usamples* from two *cvxprusage* calls. A similar estimate of user level process parallelism can be obtained by dividing *pru_utime* and *pru_pvtime.*

SEE ALSO

getrusage(2), wait(2), cvxwait(2)
"Accounting" chapter in the *CONVEX System Manager's Guide.*

NAME

dup, dup2 – duplicate a descriptor

SYNOPSIS

newd = dup(oldd)
int newd, oldd;

dup2(oldd, newd)
int oldd, newd;

DESCRIPTION

dup() duplicates an existing object descriptor. The argument *oldd* is a small, non-negative integer index in the per-process descriptor table. The value must be less than the size of the table, which is returned by *getdtablesize*(2). The new descriptor *newd* returned by the call is the lowest numbered descriptor that is not currently in use by the process.

The object referenced by the descriptor does not distinguish between references using *oldd* and *newd* in any way. Thus if *newd* and *oldd* are duplicate references to an open file, *read*(2), *write*(2), and *lseek*(2) calls all move a single pointer into the file. If a separate pointer into the file is desired, a different object reference to the file must be obtained by issuing an additional *open*(2) call.

In the second form of the call, the value of *newd* desired is specified. If this descriptor is already in use, the descriptor is first deallocated as if a *close*(2) call had been done first.

RETURN VALUE

The value –1 is returned if an error occurs in either call. The external variable *errno* indicates the cause of the error.

ERRORS

dup() and *dup2()* fail if:

[EBADF]	*oldd* or *newd* is not a valid active descriptor
[EMFILE]	Too many descriptors are active.

NOTES

These functions are an alternate interface to the *fcntl()* function using the F_DUPFD command. The call

dup(fd)

is equivalent to

fcntl(fd,F_DUPFD,0);

The call

dup2(fd,fd2)

is equivalent to

fcntl(fd,F_DUPFD,fd2)

except that

(1) If fd2 is less than zero or greater than OPEN_MAX, *dup2* fails with errno EBADF.

(2) If fd is valid, and equal to fd2, *dup2()* does not have the effect of closing and reopening the file.

(3) If fd is invalid, *dup2()* does not close fd2.

SEE ALSO

accept(2), open(2), close(2), fcntl(2), pipe(2), socket(2), socketpair(2), getdtablesize(2)

NAME

execve – execute a file

SYNOPSIS

execve(name, argv, envp)
char *name, *argv[], *envp[];

DESCRIPTION

execve() transforms the calling process into a new process. The new process is constructed from an ordinary file called the *new process file*. This file is either an executable object file, or a file of data for an interpreter. An executable object file consists of an identifying header, followed by pages of data representing the initial program (text) and initialized data pages. Additional pages may be specified by the header to be initialize with zero data.

An interpreter file begins with a line of the form "*#! interpreter.*" When an interpreter file is *execve()*'d, the system *execve()*'s the specified *interpreter*, giving it the name of the originally exec'd file as an argument, and shifting over the rest of the original arguments.

There can be no return from a successful *execve* because the calling core image is lost. This is the mechanism whereby different process images become active.

The argument *argv* is an array of character pointers to null-terminated character strings. These strings constitute the argument list to be made available to the new process. By convention, at least one argument must be present in this array, and the first element of this array should be the name of the executed program (i.e. the last component of *name*).

The argument *envp* is also an array of character pointers to null-terminated strings. These strings pass information to the new process that are not directly arguments to the command.

Descriptors open in the calling process remain open in the new process, except for those that have their close-on-exec flag is set. Descriptors which remain open are unaffected by *execve()*.

Ignored signals remain ignored across an *execve()*, but signals that are caught are reset to their default values. The signal stack is reset to an undefined state.

Each process has *real* user and group IDs and *effective* user and group IDs. The *real* ID identifies the person using the system; the *effective* ID determines his access privileges. *execve()* changes the effective user and group ID to the owner of the executed file if the file has the "set-user-ID" or "set-group-ID" modes. The *real* user ID is not affected.

The *saved-set-user-ID* and *saved-set-group-ID* are saved for use by the *setuid*(2) and *setgid*(2) functions.

The new process also inherits the following attributes from the calling process:

process ID	see *getpid*(2)
parent process ID	see *getppid*(2)
process group ID	see *getpgrp*(2)
session membership	see *setsid*
real user id	see *setuid*
real group id	see *setgid*
access groups	see *getgroups*(2)
working directory	see *chdir*(2)
root directory	see *chroot*(2)
control terminal	see *tty*(4)
resource usages	see *getrusage*(2)
interval timers	see *getitimer*(2)
resource limits	see *getrlimit*(2)
file mode mask	see *umask*(2)
signal mask	see *sigprocmask*(2)
pending signals	see *signal*(2)

When the executed program begins, it is called as follows:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the number of elements in *argv* (the "arg count") and *argv* is the array of character pointers to the arguments themselves.

envp is a pointer to an array of strings that constitute the *environment* of the process. A pointer to this array is also stored in the global variable "environ." Each string consists of a name, an "=," and a null-terminated value. The array of pointers is terminated by a null pointer. The shell *sh*(1) passes an environment entry for each global shell variable defined when the program is called. See *environ*(7) for some conventionally used names.

RETURN VALUE

If *execve()* returns to the calling process, an error has occurred; the return value will be –1 and the global variable *errno* will contain an error code.

ERRORS

execve() will fail and return to the calling process if one or more of the following are true:

[E2BIG]	The number of bytes in the argument list and environment is greater than ARG_MAX.
[ENAMETOOLONG]	The *path* argument is longer than PATH_MAX.
[ENOENT]	One or more components of the new process file's path name do not exist.
[ENOTDIR]	A component of the new process file is not a directory.
[EACCES]	Search permission is denied for a directory listed in the new process file's path prefix.
[EACCES]	The new process file is not an ordinary file.
[EACCES]	The new process file mode denies execute permission.
[ENOEXEC]	The new process file has the appropriate access permission, but has an invalid magic number in its header.
[EPERM]	The file is on a file system mounted with the "-o nosuid" option (see *mount(8)*) and the file has the setuid or setgid permission bits on.
[ETXTBSY]	The new process file is a pure procedure (shared text) file that is currently open for writing or reading by some process.
[ENOMEM]	The new process requires more virtual memory than is allowed by the imposed maximum (*getrlimit*(2)).
[EFAULT]	The new process file is not as long as indicated by the size values in its header.
[EFAULT]	*path*, *argv*, or *envp* point to an illegal address.

BACKWARD COMPATIBILITY

In previous versions of the operating system, a non-root owned, *setuid* program retained all the additional privledges of "root" when executed by "root."

SEE ALSO

exit(2), fork(2), close(2), sigvec(2), execl(3), a.out(5), environ(7)

NAME

_exit – terminate a process

SYNOPSIS

_exit(status)
int status;

DESCRIPTION

_exit() terminates a process with the following consequences:

All of the descriptors open in the calling process are closed.

If the parent process of the calling process is executing a *wait()*, or is interested in the SIGCHLD signal, then it is notified of the calling process's termination, and the low-order eight bits of *status* are made available to it.

The parent process ID of all of the calling process's existing child processes are also set to 1. This means that the initialization process inherits each of these processes as well.

Most C programs call the library routine *exit*(3) which performs cleanup actions in the standard I/O library before calling *_exit()*.

RETURN VALUE

This call never returns.

SEE ALSO

fork(2), wait(2), exit(3)

NAME

exportfs – set export characteristics of file/directory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/vfs.h>
#include <nfs/export.h>

exportfs(path, ex)
char *path;
struct export *ex;
```

DESCRIPTION

exportfs sets characteristics pertaining to NFS exported files and directories. See *exportfs*(8) for more information on the types of options that can be specified.

This call is limited to the superuser.

RETURN VALUE

exportfs returns zero if successful; -1 if not.

ERRORS

[EPERM] The effective uid of the caller is not superuser.

[ELOOP] Too many levels of symbolic links were encounted.

[EINVAL] The *ex_flags* or *ex_auth* specified are invalid.

[ENOENT] *path* does not exist.

[EFAULT] *path* or *ex* evaluate to an address outside the process's allocated address space.

[ENAMETOOLONG] *path* was longer than the maximum file name length.

SEE ALSO

mountd(8C), exportfs(8)

NOTES

exportfs is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

faillog – enable or disable logging of failed file accesses

SYNOPSIS

faillog(file)
char *file;

DESCRIPTION

The system is prepared to write a record in a log file each time a process attempts to access a file for which it does not have permission. This call, with a null-terminated string naming the existing file */usr/adm/failure_log* as the argument, turns on logging; it appends a record to */usr/adm/failure_log* for each failed access. The *faillog* system call with an argument of 0 turns off failure logging.

The log file format is shown in *failure_log*(5).

If *faillog* is invoked when logging is already turned on, the failure log switches to the new file. There is no interruption in logging when this occurs. All failures are recorded in one of the log files.

This call is permitted only to the superuser.

NOTES

Failed file access logging is automatically disabled when the file system the log file resides on becomes 98% full; it is enabled again when the file system becomes less than 96% full. The percentages may be changed using the *sysgen* tunable parameters LOG_SUSPEND and LOG_RESUME.

When the Network File System (NFS) is used, failed accesses to files residing on the NFS server are logged on the server, not the client. See the "System Generation" chapter in the *CONVEX System Manager's Guide* for more information.

RETURN VALUE

On error, –1 is returned. The file must exist and the call may be exercised only by the superuser.

ERRORS

faillog will fail if one of the following is true:

[EPERM]	The caller is not the super-user.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EISDIR]	The named file is a directory.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*File* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EACCES]	The file is a character or block special file.

SEE ALSO

failure_log(5), faillogon(8), faillogpr(8).
"System Protection" and "System Generation" chapters in the *CONVEX System Manager's Guide*

NAME

fchmod – change mode of file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
fchmod(fd, mode)
int fd;
mode_t mode;
```

DESCRIPTION

The file referenced by the descriptor *fd* has its mode changed to *mode*. Modes are constructed by *or*ing together some combination of the following:

S_ISUID	04000	set effective user ID on execution
S_ISGID	02000	set effective group ID on execution
_S_ISVTX	01000	see *sticky*(8)
S_IRUSR	00400	read by owner
S_IWUSR	00200	write by owner
S_IXUSR	00100	execute (search on directory) by owner
S_IRWXG	00070	read, write, execute (search) by group
S_IRWXO	00007	read, write, execute (search) by others

Only the owner of a file (or the super-user) may change the mode.

Writing or changing the owner of a file turns off the set-user-ID and set-group-ID bits. This makes the system somewhat more secure by protecting set-user-ID (set-group-ID) files from remaining set-user-ID (set-group-ID) if they are modified, at the expense of a degree of compatibility.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

fchmod() will fail if:

[EBADF]	The descriptor is not valid.
[EINVAL]	*fd* refers to a socket, not to a file.
[EROFS]	The file resides on a read-only file system.

SEE ALSO

chmod(2), open(2), chown(2)

NAME

fchown – change owner and group of file by descriptor

SYNOPSIS

#include <sys/types.h>

fchown(fd, owner, group)
int fd;
uid_t owner;
gid_t group;

DESCRIPTION

The file which is referenced by *fd* has its *owner* and *group* changed as specified. Only the super-user may change the owner of the file, because if users were able to give files away, they could defeat the file-space accounting procedures. The owner of the file may change the group to a group of which he is a member.

On some systems, *chown()* clears the set-user-ID and set-group-ID bits on the file to prevent accidental creation of set-user-ID and set-group-ID programs owned by the super-user.

fchown() is particularly useful when used in conjunction with the file locking primitives.

RETURN VALUE

Zero is returned if the operation was successful; –1 is returned if an error occurs, with a more specific error code being placed in the global variable *errno*.

ERRORS

fchown will fail if:

[EBADF]	*fd* does not refer to a valid descriptor.
[EINVAL]	*fd* refers to a socket, not a file.
[EPERM]	The effective user ID is not the super-user and either the file owner was being changed or the effective user ID does not match the file owner or the new group is not a member of the processes group list.
[EROFS]	The named file resides on a read-only file system.
[EIO]	An I/O error occurred while reading from or writing to the file system.

CONVEX EXTENSIONS

A value of _SAME_UID passed for owner or _SAME_GID for group results in the current owner or group remaining unchanged.

BACKWARD COMPATIBILITY

The *owner* and *group* parameters were formerly declared to be type int, although the high word was unusable.

SEE ALSO

chown(2), chown(8), chgrp(1), chmod(2), flock(2)

NAME

fcntl – file control

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

res = fcntl(fd, cmd, arg)
int res;
int fd, cmd, arg;
```

DESCRIPTION

fcntl() provides for control over descriptors. The argument *fd* is a descriptor to be operated on by *cmd* as follows:

F_DUPFD Return a new descriptor as follows:

Lowest numbered available descriptor greater than or equal to *arg*.
Same object references as the original descriptor.
New descriptor shares the same file pointer if the object was a file.
Same access mode (read, write, or read/write).
Same file status flags (i.e., both file descriptors share the same file status flags).
The close-on-exec flag associated with the new file descriptor is set to remain open across *execv*(2) system calls.

F_GETFD Get the flags associated with the file descriptor *fd*. Currently, FD_CLOEXEC is the only flag. If FD_CLOEXEC is not set, then the file will remain open across *exec*; if FD_CLOEXEC is set, then the file will be closed upon execution of *exec*.

F_SETFD Set the flags associated with *fd* to the argument *arg* .

F_GETFL Get descriptor status flags, as described below.

F_SETFL Set descriptor status flags.

F_GETLK Get a description of the first lock which would block the lock specified in the *flock* structure pointed to by *arg*. The information retrieved overwrites the information in the *flock* structure. The starting offset in the *flock* structure, *l_whence*, will always be SEEK_SET, with the *l_start* and *l_len* fields set accordingly. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except for the lock type which will be set to F_UNLCK.

F_SETLK Set or clear an advisory record lock according to the *flock* structure pointed to by *arg*. F_SETLK is used to establish shared (F_RDLCK) and exclusive (F_WRLCK) locks, or to remove either type of lock (F_UNLCK). If the specified lock cannot be applied, *fcntl* will return with an error value of -1.

F_SETLKW This *cmd* is the same as F_SETLK except that if a shared or exclusive lock is blocked by other locks, the requesting process will sleep until the lock may be applied.

The flags for the F_GETFL and F_SETFL flags are:

O_NONBLOCK Nonblocking I/O; if no data is available to a *read()* call or if a write operation would block, the call returns -1 with the error EAGAIN. *CONVEX EXTENSION:* For files and filesystems under control of a

migration daemon, if this flag is set the the process will receive an *ENOSPC* error instead of blocking if there are not enough free blocks or fragments to statisfy a filesystem allocation request. Setting O_NONBLOCK on a file for migration purposes is only valid for directories and regular files; all other file types (named pipes, devices, sockets, etc) will be silently ignored.

O_APPEND Force each write to append at the end of file; corresponds to the O_APPEND flag of *open*(2).

RETURN VALUE

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only FD_CLOEXEC is defined)
F_GETFL	Value of flags and access modes
other	Value other than –1.

Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

fcntl() will fail if one or more of the following are true:

[EACCES] The *cmd* argument is F_SETLK, the type of lock (*l_type*) is a shared lock (*F_RDLCK*) or exclusive lock (*F_WRLCK*), and the segment to be locked is already exclusive-locked by some other process, or the type is exclusive and some portion of the segment is already shared-locked or exclusive-locked by some other process.

[EBADF] *fildes* is not a valid open file descriptor.

[EBADF] *cmd* is F_RDLCK and *fildes* is not open for reading.

[EBADF] *cmd* is F_SETLK or F_SETLKW, the type is F_WRLCK, and *fildes* is not open for writing.

[EMFILE] *cmd* is F_DUPFD and the maximum allowed number of file descriptors is currently open.

[EINVAL] *cmd* is F_DUPFD and *arg* is negative or greater than the maximum allowable number (see *getdtablesize*(2)).

[EINVAL] FASIO is specified on a F_SETFL (see EXTENSIONS below) *cmd* and *fd* is DTYPE_SOCKET.

[EFAULT] *cmd* is F_GETLK, F_SETLK, or F_SETLKW and *arg* points to an invalid address.

[EINVAL] *cmd* is F_GETLK, F_SETLK, or F_SETLKW and the data *arg* points to is not valid.

[EINTR] *cmd* is F_SETLKW and a signal interrupted the process while it was waiting for the lock to be granted.

[ENOLCK] *cmd* is F_SETLK or F_SETLKW and there are no more file lock entries available.

[ENETUNREACH] *cmd* is F_GETLK, F_SETLK, or F_SETLKW and there is no loopback interface configured for the INET address family.

CONVEX EXTENSIONS

CONVEX adds the following *cmd* values as extensions:

_F_GETOWN Get the process ID or process group currently receiving SIGIO and

SIGURG signals; process groups are returned as negative values. The value returned by the _F_GETOWN function of file descriptor owner.

_F_SETOWN Set the process or process group to receive _SIGIO and _SIGURG signals; process groups are specified by supplying *arg* as negative; otherwise, *arg* is interpreted as a process ID.

_F_SETBLKSIZE
Set the physical record size for block tape operations to the value of *arg*. This value defaults to 65536 when the file is opened.

CONVEX adds the following file status flags as extensions:

_FASIO Use daemon processes to accomplish asynchronous I/O. Subsequent operations on this *fd* cause the user process to proceed in parallel with the I/O transfers. Several I/O transfers may be proceeding in parallel to or from the same file (*fd*), each under the control of a separate daemon process, limited only by a system wide configuration maximum ("ps -axl" displays daemon processes as "asiodaemon".) Synchronization may be accomplished with the *asiostat*(), *select*(), or *fstat()* system calls, or by the use of _FASYNC below. Note that it is innocently assumed that all requested I/O will be performed; the *read()* or *write()* system calls will return as if all the data had been transferred, even though the transfer may be impossible. To determine the actual results, the *asiostat()* system call will return the sum of the number of bytes transferred on all asynchronous I/O requests on a given file descriptor since the last *asiostat*() call. Note that the FASIO property may be inherited by forked children, and that it is illegal to set FASIO on socket descriptors. Asynchronous I/O seems to help tape performance, but due to the existence of the buffer cache, read ahead, and write behind, sequential file transfer is already quite asynchronous, and no performance improvement may be noted. The *brk*(), *exit*(), and *fork*() system calls will wait for all asynchronous I/O invoked by the calling process to be completed before being processed. The *asiostat*(), *close*(), *fcntl*(), *flock*(), *fstat*(), *fsync*(), *ftruncate*(), and *ioctl* system calls will wait for all asynchronous I/O on the specified file descriptor (*fd*) to be completed before being processed.

_FASYNC If the _FASIO bit is set (see above), a _SIGIO signal will be sent to a process when all its outstanding asynchronous I/O (using daemon processes) has been completed. If the _FASIO bit is not set, _FASYNC enables the _SIGIO signal to be sent to the process group when I/O is possible, e.g. upon availability of data to be read. (This latter facility is available only on tty operations.)

_FNCACHE This bit requests the kernel to bypass the incore buffer cache and perform I/O transfers directly to/from user address space. As a side effect, file system read ahead and write behind are disabled. For files that are being sequentially accessed, the net result is usually lost performance. _FNCACHE might be helpful in situations where the file in question is being randomly accessed, but you should test the results with and without this option set before using it. The operating system cannot bypass the buffer cache if the transfers are not aligned on the file's blocksize boundaries because disk controllers can't start transferring in the middle of sectors. The *fstat()* system call returns the proper blocksize as *st_blksize*. Transfers should start (by seeking if necessary) at integral multiples of *st_blksize*.

NOTES

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee exclusive access (i.e., processes may still access files without using advisory locks, possibly resulting in inconsistencies).

The record locking mechanism allows two types of locks: shared locks (F_RDLCK) and exclusive locks (F_WRLCK). More than one process may hold a shared lock for a particular segment of a file at any given time, neither multiple exclusive locks nor mixed shared and exclusive locks may exist simultaneously on any segment.

In order to claim a shared lock, the descriptor must have been opened with read access. The descriptor on which an exclusive lock is being placed must have been opened with write access.

A shared lock may be *upgraded* to an exclusive lock, and vice versa, simply by specifying the appropriate lock type with a *cmd* of F_SETLK or F_SETLKW; the previous lock will be released and the new lock applied (possibly after other processes have gained and released the lock).

If the *cmd* is F_SETLKW and the requested lock cannot be claimed immediately (e.g., another process holds an exclusive lock that partially or completely overlaps the current request) then the calling process will block until the lock may be acquired. Processes blocked awaiting a lock may be awakened by signals.

Care should be taken to avoid deadlock situations in applications in which multiple processes perform blocking locks on a set of common records.

The record that is to be locked or unlocked is described by the *flock* structure, which is defined in <fcntl.h> as follows:

```
struct flock {
        short   l_type;         /* F_RDLCK, F_WRLCK, or F_UNLCK */
        short   l_whence;       /* flag to choose starting offset */
        off_t   l_start;        /* relative offset, in bytes */
        off_t   l_len;          /* length, in bytes; 0 means lock to EOF */
        pid_t   l_pid;          /* returned with F_GETLK */
};
```

The *flock* structure describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), and size (*l_len*) of the segment of the file to be affected. *l_whence* must be set to SEEK_SET, SEEK_CUR, or SEEK_END to indicate that the relative offset will be measured from the start of the file, current position, or end-of-file, respectively. The process ID field (*l_pid*) is only used with the F_GETLK *cmd* to return the description of a lock held by another process.

Locks may start and extend beyond the current end-of-file, but must not be negative relative to the beginning of the file. A lock may be set to always extend to the end-of-file by setting *l_len* to zero. If such a lock also has *l_whence* and *l_start* set to zero, the entire file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments at either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take affect. All locks associated with a file for a given process are removed when the file is closed or the process terminates. Locks are not inherited by the child process in a *fork*(2) system call.

In order to maintain consistency in the network case, data must not be cached on client machines. For this reason, file buffering for an NFS file is turned off when the first lock is attempted on the file. Buffering will remain off as long as the file is open. Programs that do I/O buffering in the user address space, however, may have inconsistent results (the standard I/O package, for instance, is a common source of unexpected buffering).

The advisory record locking capabilities of *fcntl* are implemented throughout the network by the network lock daemon; see *lockd*(8C). If the file server crashes and is rebooted, the lock daemon will attempt to recover all locks that were associated with that server. If a lock cannot be

reclaimed, the process that held the lock will be issued a SIGLOST signal.

BUGS

The asynchronous I/O facilities of FNDELAY and FASYNC are currently available only for tty and socket operations.

File locks obtained through the *fcntl* mechanism do not interact in any way with those acquired via *flock*(2). They do, however, work correctly with the exclusive locks claimed by *lockf*(3).

F_GETLK returns F_UNLCK if the requesting process holds the specified lock. Thus, there is no way for a process to determine if it is still holding a specific lock after catching a SIGLOST signal.

In a network environment, the value of *l_pid* returned by F_GETLK is next to useless.

Setting FNDELAY for regular files (for migration purposes) affects all open instances of the file. FNDELAY should only be used by cooperating processes so that they do not block when accessing a migrated file.

SEE ALSO

close(2), dup(2), execve(2), getdtablesize(2), open(2), sigvec(2), lockf(3), lockd(8C), ifconfig(8), getpattr(2)

NAME

flock – apply or remove an advisory lock on an open file

SYNOPSIS

```
#include <sys/file.h>

#define     LOCK_SH     1     /* shared lock */
#define     LOCK_EX     2     /* exclusive lock */
#define     LOCK_NB     4     /* don't block when locking */
#define     LOCK_UN     8     /* unlock */

flock(fd, operation)
int fd, operation;
```

DESCRIPTION

Flock applies or removes an *advisory* lock on the file associated with the file descriptor *fd*. A lock is applied by specifying an *operation* parameter which is the inclusive or of either LOCK_SH or LOCK_EX and, possibly, LOCK_NB. To unlock an existing lock *operation* should be LOCK_UN.

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee consistency (i.e. processes may still access files without using advisory locks possibly resulting in inconsistencies).

The locking mechanism allows two types of locks: *shared* locks and *exclusive* locks. At any time multiple shared locks may be applied to a file, but at no time are multiple exclusive, or both shared and exclusive, locks allowed simultaneously on a file.

A shared lock may be *upgraded* to an exclusive lock, and vice versa, simply by specifying the appropriate lock type; this results in the previous lock being released and the new lock applied (possibly after other processes have gained and released the lock).

Requesting a lock on an object which is already locked normally causes the caller to be blocked until the lock may be acquired. If LOCK_NB is included in *operation*, then this will not happen; instead the call will fail and the error EWOULDBLOCK will be returned.

NOTES

Locks are on files, not file descriptors. That is, file descriptors duplicated through *dup*(2) or *fork*(2) do not result in multiple instances of a lock, but rather multiple references to a single lock. If a process holding a lock on a file forks and the child explicitly unlocks the file, the parent will lose its lock.

Processes blocked awaiting a lock may be awakened by signals.

BUGS

File locks obtained through the *flock*(2) mechanism do not interact in any way with those acquired via *fcntl*(2) or *lockf*(3).

RETURN VALUE

Zero is returned if the operation was successful; on an error a –1 is returned and an error code is left in the global location *errno*.

ERRORS

The *flock* call fails if:

[EWOULDBLOCK]	The file is locked and the LOCK_NB option was specified.
[EBADF]	The argument *fd* is an invalid descriptor.
[EINVAL]	The argument *fd* refers to an object other than a file.

SEE ALSO

open(2), close(2), dup(2), execve(2), fork(2)

NAME

fork – create a new process

SYNOPSIS

```
#include <sys/types.h>
        pid_t pid;
        pid = fork();
```

DESCRIPTION

fork() causes creation of a new process. The new process (child process) is an exact copy of the calling process except for the following:

The child process has a unique process ID.

The child process has a different parent process ID (i.e., the process ID of the parent process).

The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects. For instance, file pointers in file objects are shared between the child and the parent so that a *lseek*(2) on a descriptor in the child process can affect a subsequent *read* or *write* by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.

File locks set by the parent are not inherited by the child.

The resource utilization of a child process is set to 0; see *setrlimit*(2). (This includes *tms_utime, tms_stime, tms_cutime,* and *tms_cstime.*)

Pending alarms are cleared for the child process.

The set of signals pending for the child process is initialized to the empty set.

RETURN VALUE

Upon successful completion, *fork()* returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of –1 is returned to the parent process, no child process is created, and the global variable *errno* is set to indicate the error.

ERRORS

fork() will fail and no child process will be created if one or more of the following are true:

[EAGAIN] The system-imposed limit {PROC_MAX} on the total number of processes under execution would be exceeded.

[EAGAIN] The system-imposed limit {KID_MAX} on the total number of processes under execution by a single user would be exceeded.

[EDEADLK] The calling process is multi-threaded.

SEE ALSO

execve(2), wait(2)

NAME

fsync – synchronize a file's in-memory state with that on disk

SYNOPSIS

fsync(fd)
int fd;

DESCRIPTION

Fsync causes all modified data and attributes of *fd* to be moved to a permanent storage device. This normally results in all in-memory modified copies of buffers for the associated file to be written to a disk.

Fsync should be used by programs which require a file to be in a known state; for example in building a simple transaction facility.

RETURN VALUE

A 0 value is returned on success. A –1 value indicates an error.

ERRORS

The *fsync* fails if:

[EBADF] *Fd* is not a valid descriptor.

[EINVAL] *Fd* refers to a socket, not to a file.

SEE ALSO

sync(2), sync(8), update(8)

BUGS

The current implementation of this call is expensive for large files.

NAME

getaid – get a process' activity ID

SYNOPSIS

aid = getaid(pid)
int aid, pid;

DESCRIPTION

Getaid returns the activity ID of the process whose ID is *pid*. If *getaid* fails, the value **–1** is returned and *errno* is set to indicate the error. **–1** is not a legitimate value for a process' activity ID; any other 32-bit value is.

ERRORS

[ESRCH] The process specified does not exist.

SEE ALSO

setaid(2),
"Accounting" chapter in the *CONVEX System Manager's Guide*.

NAME

getdirentries – gets directory entries in a filesystem independent format

SYNOPSIS

```
#include <sys/dir.h>

cc = getdirentries(fd, buf, nbytes, basep)
int cc, fd;
char *buf;
int nbytes;
long *basep
```

DESCRIPTION

getdirentries attempts to put directory entries from the directory referenced by the file descriptor *fd* into the buffer pointed to by *buf*, in a filesystem independent format. Up to *nbytes* of data will be transferred. *nbytes* must be greater than or equal to the block size associated with the file, see *stat(2)*. Sizes less than this may cause errors on certain filesystems.

The data in the buffer is a series of *direct* structures each containing the following entries:

```
unsigned long                       d_fileno;
unsigned short                      d_reclen;
unsigned short                      d_namlen;
char        d_name[MAXNAMELEN + 1]; /* see below */
```

The *d_fileno* entry is a number which is unique for each distinct file in the filesystem. Files that are linked by hard links (see *link(2)*) have the same *d_fileno*. The *d_reclen* entry is the length, in bytes, of the directory record. The *d_name* entry contains a null terminated file name. The *d_namlen* entry specifies the length of the file name. Thus the actual size of *d_name* may vary from 2 to MAXNAMELEN + 1.

The structures are not necessarily tightly packed. The *d_reclen* entry may be used as an offset from the beginning of a *direct* structure to the next structure, if any.

Upon return, the actual number of bytes transferred is returned. The current position pointer associated with *fd* is set to point to the next block of entries. The pointer is not necessarily incremented by the number of bytes returned by *getdirentries*. If the value returned is zero, the end of the directory has been reached. The current position pointer may be set and retrieved by *lseek(2)*. *getdirentries* writes the position of the block read into the location pointed to by *basep*. It is not safe to set the current position pointer to any value other than a value previously returned by *lseek(2)* or a value previously returned in the location pointed to by *basep* or zero.

RETURN VALUE

If successful, the number of bytes actually transferred is returned. Otherwise, a –1 is returned and the global variable *errno* is set to indicate the error.

SEE ALSO

open(2), lseek(2)

ERRORS

getdirentries will fail if one or more of the following are true:

[EBADF]	*fd* is not a valid file descriptor open for reading.
[EFAULT]	Either *buf* or *basep* point outside the allocated address space.
[EINTR]	A read from a slow device was interrupted before any data arrived by the delivery of a signal.
[EIO]	An I/O error occurred while reading from or writing to the file system.

NAME

getdomainname, setdomainname – get/set name of current domain

SYNOPSIS

getdomainname(name, namelen)
char *name;
int namelen;

setdomainname(name, namelen)
char *name;
int namelen;

DESCRIPTION

getdomainname returns the name of the domain for the current processor, as previously set by *setdomainname*. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

setdomainname sets the domain of the host machine to be *name*, which has length *namelen*. This call is restricted to the superuser and is normally used only when the system is bootstrapped.

The purpose of domains is to enable two distinct networks that may have host names in common to merge. Each network would be distinguished by having a different domain name. At the current time, only the yellow pages service makes use of domains.

RETURN VALUE

If the call succeeds a value of 0 is returned. If the call fails, then a value of –1 is returned and an error code is placed in the global location *errno*.

ERRORS

The following errors may be returned by these calls:

[EFAULT]	The *name* parameter gave an invalid address.
[EINVAL]	The *namelen* parameter is unreasonable. Currently, there is a limit of 63 characters for the domainname.
[EPERM]	The caller was not the superuser. This error only applies to *setdomainname*.

BUGS

Domain names are limited to 63 characters.

NAME

getdtablesize – get descriptor table size

SYNOPSIS

nds = getdtablesize()
int nds;

DESCRIPTION

Each process has a fixed size descriptor table which is guaranteed to have at least 20 slots. The entries in the descriptor table are numbered with small integers starting at 0. The call *getdtablesize* returns the size of this table.

SEE ALSO

close(2), dup(2), open(2)

NAME

getfh – convert file descriptor to file handle

SYNOPSIS

```
#include <errno.h>
#include <rpc/rpc.h>
#include <sys/time.h>
#include <nfs/nfs.h>

getfh(path, fhp)
char *path;
fhandle_t *fhp;
```

DESCRIPTION

getfh returns an NFS file handle for the file *path*. The file handle is stored in a user buffer pointed to by *fhp*. This file handle may be used in the NFS remote file system protocol.

RETURN VALUE

getfh returns zero if successful; -1 if not.

ERRORS

[EPERM]	The effective uid of the caller is not superuser.
[ELOOP]	Too many levels of symbolic links were encounted.
[ENOENT]	*path* does not exist.
[EREMOTE]	*fd* referred to an open file on an NFS partition.
[EFAULT]	*path* or *fhp* evaluate to an address outside the process's allocated address space.
[ENAMETOOLONG]	*path* was longer than the maximum file name length.

SEE ALSO

mountd(8C)

NOTES

getfh is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

getgid, getegid – get group identity

SYNOPSIS

```
#include <sys/types.h>

gid_t gid;
gid = getgid()
#include <sys/types.h>

gid_t egid;
egid = getegid()
```

DESCRIPTION

getgid() returns the real group ID of the current process, *getegid* the effective group ID.

The real group ID is specified at login time.

The effective group ID is more transient and determines additional access permission during execution of a "set-group-ID" process. *getgid()* is most useful for such processes.

RETURNS

These functions are always successful; there is no return value which indicates an error.

SEE ALSO

getuid(2), setregid(2), setuid(3)

NAME

getgroups – get group access list

SYNOPSIS

```
#include <sys/types.h>
#include <limits.h>
#include <unistd.h>

ngrps = getgroups(gsize, gidset)
int ngrps, gsize;
gid_t *gidset;
```

DESCRIPTION

getgroups() gets the current group access list of the user process and stores it in the array *gidset*. The parameter *gsize* indicates the number of entries that may be placed in *gidset* . The return value is the number of entries in *gidset* that were filled.

As a special case, if *gsize* is zero, the number of groups which would be placed in *gidset* is returned without assigning into *gidset*.

No more than NGROUPS_MAX, as defined in <limits.h>, will ever be returned.

RETURN VALUE

A value of –1 indicates that an error occurred, and the error code is stored in the global variable *errno* .

BACKWARD COMPATIBILITY

In previous versions of the operating system, the array filled in by *getgroups()* was of type *int*; it is now of type *gid_t* .

CONVEX EXTENSIONS

For backward compatibility, CONVEX adds:

```
#include <sys/types.h>
#include <unistd.h>

ngrps = cvx$int_getgroups(gsize, intgidset)
int ngrps, gsize;
int *intgidset.
```

Usage is otherwise as for *getgroups()*.

ERRORS

The possible errors for *getgroups()* are:

[EFAULT]	The argument *gidset* is an invalid address.
[EINVAL]	The size of *gidset* as specified by *gsize* is not zero and is too small to accommodate the entire group access list.

SEE ALSO

setgroups(2), initgroups(3x)

NAME

gethostid, sethostid – get/set unique identifier of current host

SYNOPSIS

hostid = gethostid()
int hostid;

sethostid(hostid)
int hostid;

DESCRIPTION

Sethostid establishes a 32-bit identifier for the current processor which is intended to be unique among all UNIX systems in existence. This is normally a DARPA Internet address for the local machine. This call is allowed only to the super-user and is normally performed at boot time.

Gethostid returns the 32-bit identifier for the current processor.

SEE ALSO

hostid(1), gethostname(2)

BUGS

32 bits for the identifier is too small.

NAME

gethostname, sethostname – get/set name of current host

SYNOPSIS

gethostname(name, namelen)
char *name;
int namelen;

sethostname(name, namelen)
char *name;
int namelen;

DESCRIPTION

Gethostname returns the standard host name for the current processor, as previously set by *sethostname*. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

Sethostname sets the name of the host machine to be *name*, which has length *namelen*. This call is restricted to the superuser and is normally used only when the system is bootstrapped.

RETURN VALUE

If the call succeeds, a value of **0** is returned. If the call fails, then a value of **–1** is returned and an error code is placed in the global location *errno*.

ERRORS

The following errors may be returned by these calls:

[EINVAL]	The *namelen* parameter is unreasonable. Currently, there is a limit of 64 characters for the hostname.
[EFAULT]	The *name* parameter gave an invalid address.
[EPERM]	The caller was not the superuser.

SEE ALSO

gethostid(2), hostname(1)

BUGS

Host names are limited to MAXHOSTNAMELEN (from *<sys/param.h>*) characters, currently 64.

NAME

getitimer, setitimer – get/set value of interval timer

SYNOPSIS

```
#include <sys/time.h>

#define ITIMER_REAL      0      /* real time intervals */
#define ITIMER_VIRTUAL   1      /* virtual time intervals */
#define ITIMER_PROF      2      /* user and system virtual time */

getitimer(which, value)
int which;
struct itimerval *value;

setitimer(which, value, ovalue)
int which;
struct itimerval *value, *ovalue;
```

DESCRIPTION

The system provides each process with three interval timers, defined in *<sys/time.h>*. The *getitimer* call returns the current value for the timer specified in *which*, while the *setitimer* call sets the value of a timer (optionally returning the previous value of the timer). The timer is in microsecond units.

A timer value is defined by the *itimerval* structure:

```
struct itimerval {
        struct  timeval it_interval;    /* timer interval */
        struct  timeval it_value;       /* current value */
};
```

If *it_value* is non-zero, it indicates the time to the next timer expiration. If *it_interval* is non-zero, it specifies a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 disables a timer. Setting *it_interval* to 0 causes a timer to be disabled after its next expiration (assuming *it_value* is non-zero).

Time values smaller than the resolution of the system clock are rounded up to 10 milliseconds on the CONVEX-1.

The ITIMER_REAL timer decrements in real time. A SIGALRM signal is delivered when this timer expires.

The ITIMER_VIRTUAL timer decrements in process virtual time. It runs only when the process is executing. A SIGVTALRM signal is delivered when it expires.

The ITIMER_PROF timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the ITIMER_PROF timer expires, the SIGPROF signal is delivered. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls.

NOTES

Three macros for manipulating time values are defined in *<sys/time.h>*. *timerclear* sets a time value to zero, *timerisset* tests if a time value is non-zero, and *timercmp* compares two time values (beware that >= and <= do not work with this macro).

RETURN VALUE

If the calls succeed, a value of 0 is returned. If an error occurs, the value –1 is returned, and a more precise error code is placed in the global variable *errno*.

ERRORS

The possible errors are:

[EFAULT] The *value* structure specified a bad address.

[EINVAL] A *value* structure specified a time which was too large to be handled.

SEE ALSO

sigvec(2), gettimeofday(2)

NAME

getpagesize – get system page size

SYNOPSIS

pagesize = getpagesize()
int pagesize;

DESCRIPTION

Getpagesize returns the number of bytes in a page. Page granularity is the granularity of many of the memory management calls.

The page size is a *system* page size and may not be the same as the underlying hardware page size.

SEE ALSO

brk(2), pagesize(1)

NAME

getpattr, setpattr – get/set process attributes

SYNOPSIS

```
#include <sys/resource.h>

getpattr ( pid, pattrib )
int pid;
struct pattributes *pattrib;

setpattr ( pid, pattrib )
int pid;
struct pattributes *pattrib;
```

DESCRIPTION

getpattr returns the process attributes associated with the process specified by *pid*.

setpattr will set the specified processes attributes as defined in the attributes structure.

There are currently four supported attributes;

pattr_pfixed
When set to one specifies that the process must always be scheduled with all the CPUs available for it's use.

pattr_login
When set to one specifies that the process is an initial login process. This attribute can only be changed by the superuser, and is silently ignored when attempted to be set by a non-superuser.

pattr_pvtime
When set to one specifies that the process has the process virtual timer enabled. The process virtual timer cannot be enabled for a process while it is multithreaded (multiple threads of execution).

pattr_dmonnoblock
When set to one specifies that a user wants an error when trying to access a migrated file. If clear, then the process wants transparent access to migrated files, and is willing to be blocked if necessary.

RETURN VALUE

A 0 return value indicates that the call succeeded, changing or returning the processes attributes. A return value of –1 indicates that an error occurred, and an error code is stored in the global location *errno*.

ERRORS

The possible errors are:

[EFAULT]	The address specified for *pattrib* is invalid.
[ERSCH]	The specified *pid* is not a valid process id.
[EPERM]	The caller does not have the correct permissions to access the specified *pid*.

SEE ALSO

open(2), fcntl(2)

NAME

getpeername – get name of connected peer

SYNOPSIS

```
getpeername(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

DESCRIPTION

Getpeername returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

DIAGNOSTICS

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

accept(2), bind(2), socket(2), getsockname(2)

BUGS

Names bound to sockets in the UNIX domain are inaccessible; *getpeername* returns a zero length name.

NAME

getpflags, setpflags – get/set process entry flags

SYNOPSIS

flags = getpflags(pid)
int flags;

setpflags(newflags, mask)
int newflags;
int mask;

WARNING

The *getpflags* and *setpflags* system calls have been removed from the operating system. They are replaced by *getpattr* and *setpattr*. This manual page is left for reference in converting existing applications.

DESCRIPTION

Getpflags returns the process flags associated with the process specified in *pid*.

Setpflags will set the bits indicated by *mask* to the values given in *newflags*. The exact operation is as follows:

```
proc[pid].p_flag &= ~mask;
proc[pid].p_flag |= (newflags & mask);
```

This allows the caller to specify exactly which bits they wish to modify without the necessity to first get the flags and then make the call with the properly adjusted flags. The reason that the latter course is undesirable is that some of the process flags may have changed since the flags were obtained. Only the superuser can use this system call and extreme caution should be exercised in using it at all.

RETURN VALUE

If the call succeeds, *getpflags* returns the process flags. *setpflags* returns a value of **0**. If the call fails, *both* return a value of **–1** and an error code is placed in the global location *errno*.

ERRORS

The following errors may be returned by these calls:

[EINVAL] The *pid* is not a valid process id.

[EPERM] The caller of *setpflags* was not the superuser.

SEE ALSO

getpid(2), getppid(2)

NAME

getpgrp – get process group

SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>

pid_t pgrp;
pgrp = getpgrp();
```

DESCRIPTION

The process group of the current process is returned by *getpgrp()*.

Process groups are used for distribution of signals and by the terminal driver to arbitrate requests for terminal input. Processes that are in the controlling process group are foreground and may read, while others will block with a signal if they attempt to read.

This call is used by programs such as *csh*(1) to create process groups in implementing job control. The *tcsetpgrp*(3) and *tcgetpgrp*(3) functions are used to get/set the process group of the controlling terminal.

BACKWARD COMPATIBILITY

Previous versions of the operating system specified an argument *(pid)* to *getpgrp()*. A value of zero meant the current process. This mechanism is supported only in backward compatible binary mode.

SEE ALSO

setpgrp(2), getuid(2), tty(4)

NAME

getpid, getppid – get process identification

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t pid;
pid = getpid();

pid_t ppid;
ppid = getppid()
```

DESCRIPTION

getpid() returns the process ID of the current process. Most often it is used with the host identifier *gethostid*(2) to generate uniquely-named temporary files.

getppid() returns the process ID of the parent of the current process.

RETURNS

getpid() and *getppid()* are always successful; there is no error return value.

SEE ALSO

gethostid(2)

NAME

getpriority, setpriority – get/set program scheduling priority

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>

#define PRIO_PROCESS    0       /* process */
#define PRIO_PGRP       1       /* process group */
#define PRIO_USER       2       /* user id */

prio = getpriority(which, who)
int prio, which, who;

setpriority(which, who, prio)
int which, who, prio;
```

DESCRIPTION

The scheduling priority of the process, process group, or user, as indicated by *which* and *who* is obtained with the *getpriority* call and set with the *setpriority* call. *Which* is one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER, and *who* is interpreted relative to *which* (a process identifier for PRIO_PROCESS, process group identifier for PRIO_PGRP, and a user ID for PRIO_USER).

Prio is a value in the range –64 to 64. The default priority is 0; lower priorities cause more favorable scheduling. Internally in the kernel the priority is mapped into one of 64 run queues. The kernel always schedules the process on the lowest queue that is ready to run. Priorities can be interpreted as being in one of three groups. *Fixed high priorities* are in the range of –64 to –33. Their run queue number is computed by 32+(0.5*priority), which results in queue values 0 to 15. Priorities in the range of –32 to 32 are *variable priority*, meaning their queue value depends on the amount of recent CPU utilization. Processes which use lots of CPU cycles find their queue values incrementing; thus scheduling favors interactive, I/O bound jobs. The equation for queue value is 32+(cpu/12)+(0.75*priority), resulting in a range of 8 to 56. *Fixed low priorities* are in priority range 33 to 64. Their run queue value is computed by 32+(0.5*priority), resulting in queue values ranging from 48 to 63. Since variable priority processes never get queue values higher than 56, it is possible to place a process on a priority such that it runs only if no other process wants to (and thus consumes only otherwise idle CPU cycles.) (Note that future releases of CONVEX Unix may compute internal queue priorities differently).

CAUTION – Use the extremities of the priority ranges with great care! Internal system processes such as the swapper and the page daemon run on queue values 0 to 10. The careless use of high priorities will cause unacceptable response time for other users, and may cause system deadlocks or hangs. Fixed high priorities are designed solely for real time jobs, whose response time must be deterministic. Fixed low priorities will not provide acceptable response times for interactive jobs.

The *getpriority* call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The *setpriority* call sets the priorities of all of the specified processes to the specified value. Only the superuser may lower priorities.

RETURN VALUE

Since *getpriority* can legitimately return the value **–1**, it is necessary to clear the external variable *errno* prior to the call, then check it afterward to determine if a **–1** is an error or a legitimate value. The *setpriority* call returns **0** if there is no error, or **–1** if there is.

ERRORS

Getpriority and *setpriority* may return one of the following errors:

[ESRCH] No process(es) were located using the *which* and *who* values specified.

[EINVAL] *Which* was not one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER.

In addition to the errors indicated above, *setpriority* may fail with one of the following errors returned:

[EACCES] A process was located, but neither its effective nor real user ID matched the effective user ID of the caller.

[EACCES] A non-superuser attempted to change a process priority to a negative value.

SEE ALSO

nice(1), fork(2), renice(8)

NAME

getrlimit, setrlimit – control maximum system resource consumption

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>

getrlimit(resource, rlp)
int resource;
struct rlimit *rlp;

setrlimit(resource, rlp)
int resource;
struct rlimit *rlp;
```

DESCRIPTION

Limits on the consumption of system resources by the current process and each process it creates may be obtained with the *getrlimit* call, and set with the *setrlimit* call.

The *resource* parameter is one of the following:

RLIMIT_CPU the maximum amount of cpu time (in seconds) to be used by each process.

RLIMIT_FSIZE the largest size, in bytes, of any single file which may be created.

RLIMIT_DATA the maximum size, in bytes, of the data segment for a process; this defines how far a program may extend its break with the *sbrk*(2) system call.

RLIMIT_STACK the maximum size, in bytes, of the stack segment for a process; this defines how far a program's stack segment may be extended, either automatically by the system, or explicitly by a user with the *sbrk*(2) system call.

RLIMIT_CORE the largest size, in bytes, of a *core* file which may be created.

RLIMIT_RSS the maximum size, in bytes, a process's resident set size may grow to. This imposes a limit on the amount of physical memory to be given to a process; if memory is tight, the system will prefer to take memory from processes which are exceeding their declared resident set size.

RLIMIT_CONCUR the maximum number of processors that may be allocated to any process.

A resource limit is specified as a soft limit and a hard limit. When a soft limit is exceeded a process may receive a signal (for example, if the cpu time is exceeded), but it will be allowed to continue execution until it reaches the hard limit (or modifies its resource limit). The *rlimit* structure is used to specify the hard and soft limits on a resource,

```
struct rlimit {
        int     rlim_cur;       /* current (soft) limit */
        int     rlim_max;       /* hard limit */
};
```

Only the super-user may raise the maximum limits. Other users may only alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*.

An "infinite" value for a limit is defined as RLIM_INFINITY (0x7fffffff).

Because this information is stored in the per-process information, this system call must be executed directly by the shell if it is to affect all future processes created by the shell; *limit* is thus a built-in command to *csh*(1).

The system refuses to extend the data or stack space when the limits would be exceeded in the normal way: a *break* call fails if the data space limit is reached, or the process is killed when the stack limit is reached (since the stack cannot be extended, there is no way to send a signal!).

A file i/o operation which would create a file which is too large will cause a signal SIGXFSZ to be generated, this normally terminates the process, but may be caught. When the soft cpu time limit is exceeded, a signal SIGXCPU is sent to the offending process.

RETURN VALUE

A 0 return value indicates that the call succeeded, changing or returning the resource limit. A return value of –1 indicates that an error occurred, and an error code is stored in the global location *errno*.

ERRORS

The possible errors are:

[EFAULT] The address specified for *rlp* is invalid.

[EINVAL] The limits specified to *setrlimit* contained an invalid value.

[EPERM] The limit specified to *setrlimit* would have raised the maximum limit value, and the caller is not the super-user.

SEE ALSO

csh(1), quota(2)

BUGS

There should be *limit* and *unlimit* commands in *sh*(1) as well as in *csh*.

NAME

getrusage – get information about resource utilization

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>

#define RUSAGE_SELF      0      /* calling process */
#define RUSAGE_CHILDREN  -1     /* terminated child processes */

getrusage(who, rusage)
int who;
struct rusage *rusage;
```

DESCRIPTION

Getrusage returns information describing the resources utilized by the current process, or all its terminated child processes. The *who* parameter is one of RUSAGE_SELF and RUSAGE_CHILDREN. If *rusage* is nonzero, the buffer it points to will be filled in with the following structure:

```
struct rusage {
        struct timeval ru_utime;        /* user time used */
        struct timeval ru_stime;        /* system time used */
        long    ru_maxrss;
        long    ru_ixrss;               /* integral shared memory size */
        long    ru_idrss;               /* integral unshared data size */
        long    ru_isrss;               /* integral unshared stack size */
        long    ru_minflt;              /* page reclaims */
        long    ru_majflt;              /* page faults */
        long    ru_nswap;               /* swaps */
        long    ru_inblock;             /* block input operations */
        long    ru_oublock;             /* block output operations */
        long    ru_msgsnd;              /* messages sent */
        long    ru_msgrcv;              /* messages received */
        long    ru_nsignals;            /* signals received */
        long    ru_nvcsw;               /* voluntary context switches */
        long    ru_nivcsw;              /* involuntary context switches */
        struct timeval ru_exutime;      /*exact use_time used*/
};
```

The fields are interpreted as follows:

ru_utime the total amount of time spent executing in user mode.

ru_stime the total amount of time spent in the system executing on behalf of the process(es).

ru_maxrss the maximum resident set size utilized (in kilobytes).

ru_ixrss an "integral" value indicating the amount of memory used which was also shared among other processes. This value is expressed in units of megabytes * clock-ticks-of-execution and is calculated by summing the number of shared memory pages in use each time the internal system clock ticks.

ru_idrss an integral value of the amount of unshared memory residing in the data segment of a process (expressed in units of megabytes * clock-ticks-of-execution).

ru_isrss an integral value of the amount of unshared memory residing in the stack segment of a process (expressed in units of megabytes * clock-ticks-of-execution).

ru_minflt the number of page faults serviced without any I/O activity; here I/O activity is avoided by "reclaiming" a page frame from the list of pages awaiting reallocation.

ru_majflt the number of page faults serviced which required I/0 activity.

ru_nswap the number of times a process was "swapped" out of main memory.

ru_inblock the number of times the file system had to perform block input.

ru_oublock the number of times the file system had to perform block output.

ru_msgsnd the number of *ipc* messages sent.

ru_msgrcv the number of *ipc* messages received.

ru_nsignals the number of signals delivered.

ru_nvcsw the number of times a context switch resulted due to a process voluntarily giving up the processor before its time slice was completed (usually to await availability of a resource).

ru_nivcsw the number of times a context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice.

ru_exutime the total amount of time spent executing in user mode with microsecond accuracy.

NOTES

The numbers *ru_inblock* and *ru_outblock* account only for real I/O; data supplied by the caching mechanism is charged only to the first process to read or write the data.

SEE ALSO

gettimeofday(2), wait(2),
"Accounting" chapter in the *CONVEX System Manager's Guide.*

BUGS

There is no way to obtain information about a child process which has not yet terminated.

The resident memory sizes should not depend on system clock speed.

NAME

getsockname – get socket name

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

getsockname(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

DESCRIPTION

getsockname returns the current *name* for the specified socket. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return, it contains the actual size of the name returned in bytes.

DIAGNOSTICS

A **0** is returned if the call succeeds, **–1** if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

bind(2), socket(2)

BUGS

Names bound to sockets in the UNIX domain are inaccessible; *getsockname* returns a zero length name.

NAME

getsockopt, setsockopt – get and set options on sockets

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

getsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int *optlen;

setsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int optlen;
```

DESCRIPTION

Getsockopt and *setsockopt* manipulate *options* associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the "socket" level, *level* is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP; see *getprotoent*(3N).

The parameters *optval* and *optlen* are used to access option values for *setsockopt*. For *getsockopt* they identify a buffer in which the value for the requested option(s) are to be returned. For *getsockopt*, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is returned, *optval* may be supplied as 0.

Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file *<sys/socket.h>* contains definitions for "socket" level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section (4P).

Most socket-level options take an *int* parameter for *optval*. For *setsockopt*, the parameter should non-zero to enable a boolean option, or zero if the option is to be disabled. SO_LINGER uses a *struct linger* parameter, defined in *<sys/socket.h>*, which specifies the desired state of the option and the linger interval (see below).

The following options are recognized at the socket level. Except as noted, each may be examined with *getsockopt* and set with *setsockopt*.

SO_DEBUG	toggle recording of debugging information
SO_REUSEADDR	toggle local address reuse
SO_KEEPALIVE	toggle keep connections alive
SO_DONTROUTE	toggle routing bypass for outgoing messages
SO_LINGER	linger on close if data present
SO_BROADCAST	toggle permission to transmit broadcast messages
SO_OOBINLINE	toggle reception of out-of-band data in band
SO_SNDBUF	set buffer size for output
SO_RCVBUF	set buffer size for input
SO_TYPE	get the type of the socket (get only)
SO_ERROR	get and clear error on the socket (get only)

SO_DEBUG enables debugging in the underlying protocol modules. SO_REUSEADDR indicates that the rules used in validating addresses supplied in a *bind*(2) call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a SIGPIPE signal. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are

directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messages are queued on socket and a *close*(2) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the *close* attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the *setsockopt* call when SO_LINGER is requested). If SO_LINGER is disabled and a *close* is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with *recv* or *read* calls without the MSG_OOB flag. SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values. Finally, SO_TYPE and SO_ERROR are options used only with *setsockopt*. SO_TYPE returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

RETURN VALUE

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOPROTOOPT]	The option is unknown at the level indicated.
[EFAULT]	The address pointed to by *optval* is not in a valid part of the process address space. For *getsockopt*, this error may also be returned if *optlen* is not in a valid part of the process address space.
[EINVAL]	*optval* is not a valid pointer.
[EINVAL]	*level* is negative.

SEE ALSO

ioctl(2), socket(2), getprotoent(3N)

BUGS

Several of the socket options should be handled at lower levels of the system.

NAME

getsysinfo – get system information

SYNOPSIS

```
#include      <sys/sysinfo.h>
getsysinfo(SYSINFO_SIZE, sysinfo)
struct        system_information  *sysinfo;
```

DESCRIPTION

Getsysinfo returns information describing the configuration of the CONVEX Computer System on which you are running. This system call is specific to the CONVEX implementation of UNIX.

SYSINFO_SIZE is defined in *<sys/sysinfo.h>* as the size of the *system_information* structure. This value is used as a protection mechanism when more information is provided in the *system_information* structure. If older software is run against a newer, larger version of *struct system_information,* the **SYSINFO_SIZE** value supplied will be the number of bytes of *system_information* placed in *sysinfo.*

The buffer *sysinfo* points to will be filled in with the following structure:

```
struct system_information {
        unsigned short  system_sn;    /* serial number of system */
        unsigned char   cpu_type;     /* type of CPUs in complex */
        unsigned char   cpu_count;    /* # CPUs in complex       */
        long long       flags;        /* flags                   */
};
```

The fields are interpreted as follows. More complete definitions of the fields may be found in */usr/include/sys/sysinfo.h.*

system_sn	serial number of the system on which the program is running.
cpu_type	indicates the CPU type of the CPU(s) in the complex
cpu_count	indicates the number of CPU(s) in the complex
flags	various flags indicating hardware and software capabilities.

RETURN VALUE

If the call succeeds a value of 0 is returned. If the call fails, then a value of –1 is returned and an error code is placed in the global location *errno.*

ERRORS

The following error code may be set in *errno* :

[EINVAL]	SYSINFO_SIZE is less than or equal to zero.

SEE ALSO

gethostid(2), gethostname(2), uname(2)

NAME

gettimeofday, settimeofday – get/set date and time

SYNOPSIS

```
#include <sys/time.h>

gettimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;

settimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;
```

DESCRIPTION

Gettimeofday returns the system's notion of the current Greenwich time and the current time zone. Time returned is expressed relative in seconds and microseconds since midnight January 1, 1970.

The structures pointed to by *tp* and *tzp* are defined in *<usr/include/sys/time.h>* as:

```
struct timeval {
        long    tv_sec;         /* seconds since Jan. 1, 1970 */
        long    tv_usec;        /* and microseconds */
};

struct timezone {
        int     tz_minuteswest; /* of Greenwich */
        int     tz_dsttime;     /* type of dst correction to apply */
};
```

The *timezone* structure indicates the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

Only the super-user may set the time of day.

RETURN

A 0 return value indicates that the call succeeded. A –1 return value indicates an error occurred, and in this case an error code is stored into the global variable *errno*.

ERRORS

The following error codes may be set in *errno*:

[EFAULT] An argument address referenced invalid memory.

[EPERM] A user other than the super-user attempted to set the time.

SEE ALSO

date(1), ctime(3)

NAME

getuid, geteuid – get user identity

SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>

uid_t uid;
uid = getuid()

#include <unistd.h>
#include <sys/types.h>

uid_t euid;
euid = geteuid()
```

DESCRIPTION

getuid() returns the real user ID of the current process, and *geteuid()* returns the effective user ID.

The real user ID identifies the person who is logged in. The effective user ID gives the process additional permissions during execution of "set-user-ID" mode processes, that use *getuid()* to determine the real-user-id of the process which invoked them.

RETURNS

These functions are always successful; there is no return value which indicates an error.

SEE ALSO

getgid(2), setreuid(2)

NAME

ioctl – control device

SYNOPSIS

#include <sys/ioctl.h>

ioctl(d, request, argp)
int d, request;
char *argp;

DESCRIPTION

Ioctl performs a variety of functions on open descriptors. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with *ioctl* requests. The write-ups of various devices in section 4 discuss how *ioctl* applies to them.

An ioctl *request* has encoded in it whether the argument is an "in" parameter or "out" parameter, and the size of the argument *argp* in bytes. Macros and defines used in specifying an ioctl *request* are located in the file *<sys/ioctl.h>*.

RETURN VALUE

If an error has occurred, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

Ioctl will fail if one or more of the following are true:

[EBADF] *D* is not a valid descriptor.

[ENOTTY] *D* is not associated with a character special device.

[ENOTTY] The specified request does not apply to the kind of object which the descriptor *d* references.

[EINVAL] *Request* or *argp* is not valid.

[EPERM] Only superusers may perform the specified ioctl.

SEE ALSO

execve(2), fcntl(2), ta(4), tty(4), intro(4N)

NAME

kill – send signal to a process

SYNOPSIS

int kill(pid, sig)
pid_t pid;
int sig;

DESCRIPTION

kill() sends the signal *sig* to a process specified by *pid*. *sig* may be either one of the signals specified in *sigvec*(2) or 0, in which case error checking is performed, but no signal is actually sent. This can be used to check the validity of *pid*.

For a process to have permission to send a signal, the real or effective user ID of the sending process must match the real or saved set-user-ID of the receiving process; otherwise, this call is restricted to the super-user. A single exception is the signal SIGCONT that may always be sent to any member of the same session as the sender.

If the process number is 0, the signal is sent to all other processes in the senders process group; this is a variant of *killpg*(2).

If the process number is –1 and the user is the super-user, the signal is broadcast universally except to system processes and the process sending the signal. If the process number is –1 and the user is not the super-user, the signal is broadcast universally to all processes with the same UID as the user except the process sending the signal. No error is returned if any process could be signaled.

For compatibility with System V, if the process number is negative but not –1, the signal is sent to all processes whose process group ID is equal to the absolute value of the process number. This is a variant of *killpg*(2).

Processes may send signals to themselves.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

kill will fail and no signal will be sent if any of the following occur:

[EINVAL]	*sig* is not a valid signal number.
[ESRCH]	No process can be found corresponding to that specified by *pid*.
[ESRCH]	The process ID was given as 0 but the sending process does not have a process group.
[EPERM]	The sending process is not the super-user and its real or effective user ID does not match the real or save-set-user-ID of the receiving process. When signaling a process group, this error is returned if any members of the group could not be signaled.

BACKWARD COMPATIBILITY

Previous versions of the operating system based the permission test solely on a match of the effective UID's of the sender and receiver.

SEE ALSO

kill(1), getpid(2), getpgrp(2), killpg(2), sigvec(2)

NAME

killpg – send signal to a process group

SYNOPSIS

killpg(pgrp, sig)
int pgrp, sig;

DESCRIPTION

Killpg sends the signal *sig* to the process group *pgrp*. See *sigvec*(2) for a list of signals.

The sending process and members of the process group must have the same effective user ID, otherwise this call is restricted to the super-user. As a single special case the continue signal SIGCONT may be sent to any process which is a descendant of the current process.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

Killpg will fail and no signal will be sent if any of the following occur:

[EINVAL] *Sig* is not a valid signal number.

[ESRCH] No process can be found corresponding to that specified by *pid*.

[EPERM] The sending process is not the super-user and one or more of the target processes has an effective user ID different from that of the sending process.

SEE ALSO

kill(2), getpgrp(2), sigvec(2)

NAME

limits – return or set limits or cost structures

SYNOPSIS

```
#include <sys/types.h>
#include <sys/lnode.h>
#include <sys/share.h>

limits(address, function)
struct lnode *address;
int function;
```

DESCRIPTION

This system call manipulates either a kernel limits structure or a cost structure, according to the value of *function*. Except where indicated below, *address* points to an *lnode* or an array of *lnodes*.

Function	Value	Meaning
L_ALLKN	11	All active kern_lnodes are returned.
L_ALLLIM	2	All active lnodes are returned.
L_CHNGLIM	5*	Changes limits fields in existing lnode.
L_DEADGROUP	6*	Pick up a dead lnode.
L_GETCOSTS	7	Get contents of system shconsts table.
L_MYKN	9	Get user's own kern_lnode.
L_MYLIM	0	Get user's own lnode.
L_OTHKN	10	Get kern_lnode associated with uid.
L_OTHLIM	1	Get limits associated with uid in lnode.
L_SETCOSTS	8*	Set contents of system shconsts table.
L_SETLIM	3*	Connect to a new lnode.

The starred functions in the list are super–user only. Invalid functions return an error of EINVAL.

L_ALLKN

address should point to an array of *struct kern_lnodes* defined in *<sys/lnode.h>*. The function fills in the array with *kern_lnodes* and returns the number of active *kern_lnodes* found.

L_ALLLIM

address should point to an array of *struct lnode* defined is *<sys/lnode.h>*. The function fills the array with active lnodes and returns the number of active lnodes found.

L_CHNGLIM

The uid in the passed lnode should identify the kern_lnode to be changed. Fails if the desired user doesn't exist [ESRCH].

L_DEADGROUP

Looks for a dead kern_lnode, removes it from the active list, and returns the lnode. Fails if there are no dead kern_lnodes [ESRCH].

L_GETCOSTS

address should point to a *struct shconsts* defined in *<sys/share.h>*. The parameters and statistics for the scheduling algorithm are returned in the passed shconsts.

L_MYKN

address should point to a kern_lnode. The caller's kern_lnode is returned in the passed kern_lnode structure, the function returns the number of processes attached to the caller's kern_lnode.

L_MYLIM

address should point to an lnode. The caller's lnode is returned in the passed lnode, and the

function returns the number of processes attached to the caller's lnode.

L_OTHKN
address should point to a kern_lnode defined in *<sys/lnode.h>*. The uid in the passed kern_lnode should identify the kern_lnode to be returned. Returns the number of processes attached to the kern_lnode. Fails if the desired user doesn't exist [ESRCH].

L_OTHLIM
The uid in the passed lnode should identify the lnode to be returned. Returns the number of processes attached to the lnode. Fails if the desired user doesn't exist [ESRCH].

L_SETCOSTS
address should point to a struct shconsts. The parameters for the scheduling algorithm are set from the passed shconsts structure.

L_SETLIM
Finds an existing kern_lnode with the same uid as the passed lnode, or initializes a new kern_lnode with the passed lnode, and attaches the calling process to it. All children of that process will inherit the new kern_lnode . Fails if the user's group has not been installed previously [ESRCH], or if the kernel's maximum allowed user limit would be exceeded [ETOOMANYU].

DIAGNOSTICS

Any error causes a –1 to be returned; *errno* will be set to indicate specific system call errors.

SEE ALSO

setlimits(3), lnode(5), share(5)

NOTES

limits is part of the Share scheduler. Share is an optional product; please contact your CONVEX sales representative for more information.

NAME

link – make a hard link to a file

SYNOPSIS

int link(name1, name2)
char *name1, *name2;

DESCRIPTION

A hard link to *name1* is created; the link has the name *name2*. *name1* must exist.

With hard links, both *name1* and *name2* must be in the same file system. Unless the caller is the super-user, *name1* must not be a directory. Both the old and the new *link* share equal access and rights to the underlying object.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

link() will fail and no link will be created if one or more of the following are true:

[EACCES] A component of either path prefix denies search permission.
The requested link requires writing in a directory with a mode that denies write permission.

[EEXIST] The link named by *name2* does exist.

[EINVAL] Either pathname contains a byte with the high-order bit set.

[EMLINK] The number of links to *name1* would exceed LINK_MAX for the file system containing *name1*.

[ENAMETOOLONG]
The length of *name1* or *name2* exceeds PATH_MAX for the file system, or a pathname component exceeds NAME_MAX for a file system with _POSIX_NO_TRUNC in effect.

[ENOENT] A component of either path prefix does not exist.
The file named by *name1* does not exist.
name1 or *name2* points to an empty string.

[ENOTDIR] A component of either path prefix is not a directory.

[EPERM] The file named by *name1* is a directory.

[EROFS] The requested link requires writing in a directory on a read-only file system.

[EXDEV] The link named by *name2* and the file named by *name1* are on different file systems.

[EFAULT] One of the pathnames specified is outside the process's allocated address space.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

BACKWARD COMPATIBILITY

Previous versions of the operating system allowed the empty path string as a synonym for the current directory.

Previous versions of the operating system allowed root to create hard links to directories.

SEE ALSO

symlink(2), unlink(2)

NAME

listen – listen for connections on a socket

SYNOPSIS

listen(s, backlog)
int s, backlog;

DESCRIPTION

To accept connections, a socket is first created with *socket*(2), a willingness to accept incoming connections and a queue limit for incoming connections are specified with *listen*(2), and then the connections are accepted with *accept*(2). The *listen* call applies only to sockets of type SOCK_STREAM or SOCK_SEQPACKET.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error with an indication of ECONNREFUSED, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

RETURN VALUE

A 0 return value indicates success; –1 indicates an error.

ERRORS

The call fails if:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EOPNOTSUPP]	The socket is not of a type that supports the operation *listen*.

SEE ALSO

accept(2), connect(2), socket(2)

BUGS

The *backlog* is currently limited (silently) to 5.

NAME

lseek – move read/write pointer

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(d, offset, whence)
int   d, whence;
off_t offset;
```

DESCRIPTION

The descriptor *d* refers to a file or device open for reading and/or writing. *lseek()* sets the file pointer of *d* as follows:

If *whence* is SEEK_SET, the pointer is set to *offset* bytes.

If *whence* is SEEK_CUR, the pointer is set to its current location plus *offset*.

If *whence* is SEEK_END, the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location as measured in bytes from beginning of the file is returned. Some devices are incapable of seeking. The value of the pointer associated with such a device is undefined.

RETURN VALUE

Upon successful completion, a non-negative integer, the current file pointer value, is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

lseek() will fail and the file pointer will remain unchanged if:

[EBADF]	*d* is not an open file descriptor.
[EINVAL]	*whence* is not a proper value.
[EINVAL]	The resulting file pointer would be negative.
[ESPIPE]	*d* is associated with a pipe or a socket.

NOTES

Seeking far beyond the end of a file, then writing, creates a gap or "hole," which occupies no physical space and reads as zeros.

SEE ALSO

dup(2), fseek(3s), open(2)

NAME

lstat – get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

lstat(path, buf)
char *path;
struct stat *buf;
```

DESCRIPTION

lstat obtains information about the file named by *path*. Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

lstat operates like *stat*(2) except in the case where the named file is a symbolic link, in which case *lstat* returns information about the link while *stat* returns information about the file the link references.

buf is a pointer to a *stat* structure into which information is placed concerning the file. The contents of the structure pointed to by *buf* include the following members:

```
struct stat {
        dev_t     st_dev;     /* device inode resides on */
        ino_t     st_ino;     /* this inode's number */
        mode_t    st_mode;    /* protection */
        nlink_t   st_nlink;   /* number of hard links to the file */
        uid_t     st_uid;     /* user-id of owner */
        gid_t     st_gid;     /* group-id of owner */
        off_t     st_size;    /* total size of file */
        time_t    st_atime;   /* file last access time */
        time_t    st_mtime;   /* file last modify time */
        time_t    st_ctime;   /* file last status change time */
        /* The following are CONVEX extensions */
        dev_t     st_rdev;    /* the device type, for inode that is device */
        long      st_blksize; /* optimal blocksize for file system i/o ops */
        long      st_blocks;  /* actual number of blocks allocated */
};
```

st_atime — Time when file data was last read or modified. Changed by the following system calls: *mknod*(2), *utimes*(2), *read*(2), *write*(2), and *truncate*(2). For reasons of efficiency, st_atime is not set when a directory is searched though this would be more logical.

st_mtime — Time when data was last modified. It is not set by changes of owner, group, link count, or mode. Changed by the following system calls: *mknod*(2), *truncate*(2), *utimes*(2), *write*(2).

st_ctime — Time when file status was last changed. It is set both both by writing and changing the inode. Changed by the following system calls: *chmod*(2) *chown*(2), *link*(2), *mknod*(2), *rename*(2), *unlink*(2), *utimes*(2), *write*(2), *truncate*(2).

st_mode — The attributes of st_mode may be queried with the following macros:

S_ISDIR(st.st_mode) True if a directory

S_ISCHR(st.st_mode) True if a character special device

S_ISBLK(st.st_mode) True if a block special device

S_ISREG(st.st_mode) True if a regular file

S_ISFIFO(st.st_mode) True if a pipe or fifo special file

_S_IFLNK(st.st_mode) True if a symbolic link

_S_ISSOCK(st.st_mode) True if a socket

The permissions associated with st_mode may be extracted with the S_IRWXU, S_IRWXG, and S_IRWXO masks. The set-user-ID bit is S_ISUID and the set-group-ID bit is S_ISGID. Refer to *chmod*(2) for more details. The file mode bits will contain _S_ISVTX if the sticky bit is set (see sticky(8)).

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

lstat will fail if one or more of the following are true:

[ENOENT] *path* points to an empty string.

[ENOTDIR] A component of the path prefix of *path* is not a directory.

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG]
The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters. The file referred to by *path* does not exist.

[EACCES] Search permission is denied for a component of the path prefix of *path*.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[EFAULT] *buf* or *path* points to an invalid address.

[EIO] An I/O error occurred while reading from or writing to the file system.

BACKWARD COMPATIBLITY

In previous versions of the operating system, the empty pathname string was a synonym for the current directory.

The file status information was previously expressed as follows:

```
#define  S_IFMT     0170000   /* type of file */
#define  S_IFIFO    0010000   /* fifo special */
#define  S_IFCHR    0020000   /* character special */
#define  S_IFDIR    0040000   /* directory */
#define  S_IFBLK    0060000   /* block special */
#define  S_IFREG    0100000   /* regular */
#define  S_IFLNK    0120000   /* symbolic link */
#define  S_IFSOCK   0140000   /* socket */
#define  S_ISUID    0004000   /* set user id on execution */
#define  S_ISGID    0002000   /* set group id on execution */
#define  S_ISVTX    0001000   /* see sticky(8) */
#define  S_IREAD    0000400   /* read permission, owner */
#define  S_IWRITE   0000200   /* write permission, owner */
#define  S_IEXEC    0000100   /* execute/search permission, owner */
```

The mode bits 0000070 and 0000007 encode group and others permissions (see *chmod*(2)).

SEE ALSO

chmod(2), chown(2), fstat(2), readlink(2), stat(2), utimes(2)

NAME

mkdir – make a directory file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int mkdir(path, mode)
char *path;
mode_t mode;
```

DESCRIPTION

mkdir() creates a new directory file with name *path*. The mode of the new file is initialized from *mode*.

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to that of the parent directory in which it is created.

The low-order 9 bits of mode are modified by the process's file mode creation mask. All bits set in the process's file mode creation mask are cleared.

RETURN VALUE

A 0 return value indicates success. A –1 return value indicates an error, and an error code is stored in *errno*.

ERRORS

mkdir() will fail and no directory will be created if:

[EACCES] Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the directory to be created.

[EEXIST] The named file exists.

[EINVAL] The *path* argument contains a byte with the high-order bit set.

[EMLINK] The link count of the parent directory would exceed LINK_MAX for that file system.

[ENAMETOOLONG] The length of *path* exceeds PATH_MAX for the file system.

[ENAMETOOLONG] The length of a pathname component exceeds NAME_MAX and the file system enforces _POSIX_NO_TRUNC.

[ENOENT] A component of the path prefix does not exist.
The *path* argument points to the empty string.

[ENOSPC] The file system does not contain enough space to hold the contents of the new directory or to expand the parent directory.

[ENOTDIR] A component of the path prefix is not a directory.

[EROFS] The named file resides on a read-only file system.

[EFAULT] *path* points outside the process's allocated address space.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EIO] An I/O error occurred while writing to the file system.

SEE ALSO

chmod(2), stat(2), umask(2)

NAME

mknod – make a special (character, block, or fifo) file

SYNOPSIS

mknod(path, mode, dev)
char *path;
int mode, dev;

DESCRIPTION

mknod creates a new file named by the path name pointed to by *path*. The mode of the new file (including file type bits) is initialized from *mode*. The values of the file type bits which are permitted are:

```
#define  S_IFCHR   0020000   /* character special */
#define  S_IFBLK   0060000   /* block special */
#define  S_IFREG   0100000   /* regular */
#define  S_IFIFO   0010000   /* FIFO special */
```

Values of *mode* other than those above are undefined and should not be used.

The protection part of the mode is modified by the process's mode mask; (see *umask*(2)).

The owner ID of the file is set to the effective user ID of the process. The group ID of the file is set to the group ID of the parent directory.

If mode indicates a block or character special file, *dev* is a configuration dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

mknod may be invoked only by the super-user for file types other than FIFO special.

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

mknod fails and the file mode will be unchanged if:

[ENOTDIR] A component of the path prefix of *path* is not a directory.

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG] The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.

[ENOENT] A component of the path prefix of *path* does not exist.

[EACCES] Search permission is denied for a component of the path prefix of *path*.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[EPERM] An attempt was made to create a file of type other than FIFO special and the process's effective user ID is not super-user.

[EIO] An I/O error occurred while reading from or writing to the file system.

[EISDIR] The specified *mode* would have created a directory.

[ENOSPC] The directory in which the entry for the new file is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENOSPC] There are no free inodes on the file system on which the file is being created.

[EDQUOT] The directory in which the entry for the new file is being placed cannot be

extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EDQUOT] The user's quota of inodes on the file system on which the node is being created has been exhausted.

[EROFS] The file referred to by *path* resides on a read-only file system.

[EEXIST] The file referred to by *path* exists.

[EFAULT] *path* points outside the process's allocated address space.

SEE ALSO

chmod(2), stat(2), umask(2), mknod(8)

NAME

mmap – map shared memory into your address space

SYNOPSIS

```
#include      <sys/types.h>
#include      <sys/mman.h>
caddr_t
mmap(addr, len, prot, share, fd, offset)
caddr_t addr; unsigned *len, prot, share;
int fd; off_t offset;
```

DESCRIPTION

The *mmap* system call maps shared memory into the calling process' address space. Shared memory segments may be mapped between the top of the data segment and the bottom of the stack segment. All processes which map the same shared memory segment into their address space are guaranteed to be accessing the same physical pages, even though at possibly different virtual addresses (if they are on the same machine).

The *mmap* parameters specify how the mapping is to be done. Many of them assume logical default values– see below. *addr* specifies the desired virtual address of the mapping. *len* specifies the length of the segment, which may be larger than the mapped object. The actual length of the mapped object is returned in *len*.

prot specifies the desired page protection bits in the page table entries, and is composed of the logical-OR of PROT_READ, PROT_WRITE, and PROT_EXEC.

share is composed of the logical-OR of the mapping type, and various mapping options. There are four mapping types (or domains)– MAP_FILE, MAP_ANON, MAP_THREAD, and MAP_DEVICE. The available options (described below) include MAP_FIXED, MAP_INHERIT, MAP_HASSEMAPHORE, MAP_EXTEND, MAP_SHARED, MAP_PRIVATE, MAP_FILLFD, MAP_GROWAUTO, MAP_GROWUP, MAP_GROWDOWN, and MAP_DEBUG.

fd is a Unix file descriptor, that identifies the file or character device to be mapped, or names an area of swap space that is mapped. *offset* is the the offset in the mapped object at which mapping begins. It must be a multiple of NBPG, the number of bytes per page.

In the MAP_FILE domain, the argument *fd* is the file descriptor of a regular file, which will in effect, be mapped into your virtual address space. Pages in the mapped area which are read will automatically be initialized to the contents of the file. If *share* is MAP_PRIVATE, dirty (written into) pages are replicated and made private to the writing process. Dirty, private pages are paged to the swap area. The regular file is left undisturbed. If MAP_SHARED, all processes writing into pages in the mapped area are altering the same physical pages. Dirty, shared pages are paged to the regular file irregularly, if at all.

The *msync* system call flushes dirty, shared pages to the buffer cache. Subsequent *sync* and *fsync* system calls will process the data for the file. When the last process using a shared memory segment in the MAP_FILE domain unmaps the segment, an automatic *msync* of the mapped area occurs.

It is possible to enlarge the file with write accesses (see below.) It is not necessary that the entire file be mapped into the process' address space; the *offset* parameter specifies the byte offset in the file which corresponds to the first byte in the mapped segment.

In the MAP_ANON domain, the argument *fd* is the file descriptor of a regular file. This regular file is not disturbed. Pages are initialized to zero, and paged to a temporary file in /swap if *share* is MAP_SHARED, or paged to the swap area if *share* is MAP_PRIVATE. Cooperating processes can MAP_SHARE the same segment (and so share the same temporary file), by mapping the same regular file in the MAP_ANON domain. After the last process attached to a shared memory segment in the MAP_ANON domain unmaps the segment, the contents of the segment are lost, and,

if present, the temporary file is deleted. Currently, a segment in the MAP_ANON domain cannot be enlarged after the initial mmap call. The *offset* argument should be zero for the MAP_ANON domain.

In the MAP_THREAD domain, the argument *fd* is the file descriptor of a regular file, which will in effect, be mapped into your virtual address space. Pages in the mapped area which are read will automatically be initialized to the contents of the file. If *share* is MAP_PRIVATE, dirty (written into) pages are replicated and made private for each writing thread of your process. Dirty, private pages are paged to the swap area. The regular file is left undisturbed. MAP_SHARED is not currently allowed. It is not necessary that the entire file be mapped into the process' address space; the *offset* parameter specifies the byte offset in the file which corresponds to the first byte in the mapped segment.

The MAP_DEVICE domain supports access to kernel and physical memory directly, and may in the future be used to implement device drivers with buffers mapped directly into user memory. *fd* must be a character special device that has a map entry point in its device driver. The /dev/mem and /dev/kmem devices allow the mapping of physical and kernel virtual memory into user address space. No other devices currently support the map entry point. The *offset* argument is the physical or kernel virtual address that is mapped as the first byte in the shared segment. Paging is not possible from segments in the MAP_DEVICE domain, since physical and kernel virtual memory are always "resident". Mapping with MAP_DEVICE is inherently machine dependent.

A single *fd* cannot be used to map segments in different domains simultaneously; e.g. if a process uses the file with MAP_FILE another process cannot perform a MAP_ANON on that file.

The mapping options include the type (or domain); MAP_SHARED, which specifies that segments are to be shared with other processes mapping the same segment; MAP_PRIVATE, which specifies that changes made to the segment are private to this process. MAP_EXTEND, which is used to extend or lengthen files (see below); MAP_INHERIT, which specifies that shared memory segments are to be retained across the *exec* call; MAP_HASSEMAPHORE, that specifies the shared memory segment has one or more semaphores (see *msleep*(2)); MAP_FILLFD causes anonymous memory regions to use the file descriptor passed for initialization of pages but changes are paged to swap instead of back to the file as MAP_FILE does. MAP_DEBUG, which causes the kernel to print debug information on why calls to *mmap* fail; and MAP_FIXED, that specifies the kernel may not modify *addr* as it does the mapping.

The *prot* argument is the logical-OR of PROT_READ, PROT_WRITE, and PROT_EXEC, corresponding to the ability to read, write, or execute the mapped pages. Ability to read, write, or execute the mapped segment is granted only if the file identified by *fd* grants read, write, or execute permission to the calling process.

The shared memory segment is mapped into the process' virtual address space at *addr*, or if *addr* is zero, a default address is provided by the kernel, which is the address at which the segment is mapped by other processes (so that pointers will work.) Address 0xc0000000 is provided if no other processes have *fd* mapped. *addr* cannot lie within the process' text, data, or stack segments (see below). The kernel may arbitrarily round or change the address at which the segment will be mapped unless MAP_FIXED. If MAP_FIXED is specified an error will be returned if the user's specified *addr* is unsuitable.

The *len* argument is a pointer to an argument that specifies the length of the mapped segment; it represents the largest possible file in the MAP_FILE domain, or the largest possible memory region in the MAP_ANON domain. For a memory region with the MAP_GROWAUTO attribute, len specifies the limit for the memory region to grow very much as the *stacksize* of a process is limited. A mapped file may be smaller than *len*; the actual length of a mapped file, or the size of a single page for MAP_GROWAUTO regions, is returned in *len*. If *len* is zero for a segment in the MAP_FILE domain, the length of the mapped segment will be the length of the file specified by *fd* rounded up to the nearest page boundary.

References beyond the end of file may optionally extend the file if MAP_EXTEND is set in *share* and the domain is MAP_FILE, up to the limit in the *len* argument. If MAP_EXTEND is not set, bus error signals (SIGBUS) will be generated, which normally terminate the process. MAP_EXTEND currently has no meaning in the MAP_ANON domain.

Shared segments are preserved across *fork* and *vfork*, allowing children to reference the shared memory segments owned by their parents. Shared memory segments will be inherited across *exec* if and only if the MAP_INHERIT flag is set in *share*. Shared memory segments are unmapped as a result of process termination, *exit*, or *munmap*.

BUGS

Using *read* or *write* on a file that is simultaneously mapped in the MAP_FILE domain is not prohibited; however it is not guaranteed that file system accesses and shared memory accesses will see a consistent view of the file, because of simultaneous access by the system and the mapping processes.

It is not possible to *truncate* or *ftruncate* a file while it is mapped into a shared memory segment; any attempt to do so will be return an error.

RETURN VALUE & ERRORS

mmap returns a character pointer to the first byte in the mapped segment if the mapping was successful. If unsuccessful, mmap returns a -1 with errno set as follows:

[EINVAL] *offset* is not a multiple of NBPG (the number of bytes per page), or *offset* is negative, or *fd* is not a regular file for MAP_FILE domain, or *fd* is not a character special device for MAP_DEVICE domain, or *len* is zero and the domain is not MAP_FILE, or no domain was selected in the *share* argument, or the given *fd* is already mapped under a different domain, or the shared segment overlaps with another segment (see BUGS above).

[ESPIPE] *fd* is not an inode.

[ENODEV] The character special device *fd* does not have a map entry in the MAP_DEVICE domain.

[EBADF] *fd* was not a valid file descriptor.

[EACCES] The desired page protection (PROT_READ, PROT_WRITE, and PROT_EXEC) conflict with the permissions granted by *fd*.

[ENOENT] The directory /swap does not exist.

[ENOMEM] There is insufficient swap space or memory to grant the request.

SEE ALSO

munmap(2), msleep(2), mwakeup(2), msync(2), tas(3), mremap(2)
Using C-1 Shared Memory

NAME

mount – mount file system

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mount.h>

mount(type, dir, flags, data)
int type;
char *dir;
int flags;
caddr_t data;
```

DESCRIPTION

mount attaches a file system to a directory. After a successful return, references to directory *dir* will refer to the root directory on the newly mounted file system. *dir* is a pointer to a null-terminated string containing a path name. *dir* must exist already, and must be a directory. Its old contents are inaccessible while the file system is mounted.

The *flags* argument determines whether the file system can be written on, and if set-uid execution is allowed. Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

type indicates the type of the filesystem. It must be one of the types defined in *mount.h*. *data* is a pointer to a structure which contains the type specific arguments to mount. Below is a list of the filesystem types supported and the type specific arguments to each:

MOUNT_UFS

```
struct ufs_args {
   char   *fspec;       /* block special file to mount */
   int    blkhi;        /* percentage of full disk for highwater mark */
   int    blklo;        /* percentage of full disk for lowater mark */
   long   ihi;          /* High water mark for inode allocation */
   long   ilo;          /* Lo water mark for inode allocation */
   u_int  migflags;     /* migration control flags */

};
```

MOUNT_NFS

```
#include       <nfs/nfs.h>
#include       <netinet/in.h>
struct nfs_args {
     struct sockaddr_in *addr;    /* file server address */
     fhandle_t          *fh;     /* File handle to be mounted */
     int                flags;    /* flags */
     int                wsize;    /* write size in bytes */
     int                rsize;    /* read size in bytes */
     int                timeo;    /* initial timeout in .1 secs */
     int                retrans;  /* times to retry send */
     char               *hostname; /* server's name */
};
```

RETURN VALUE

mount returns 0 if the action occurred, and –1 if one of the errors below is detected.

ERRORS

mount will fail when one of the following occurs:

[EPERM] The effective uid of the caller is not superuser.

[EINVAL] The pathname of *fspec* (MOUNT_UFS) or *dir* contains a character with the high-order bit set.

[ENOTDIR] A component of the path prefix in *fspec* (MOUNT_UFS) or *dir* is not a directory.

[ENOENT] *fspec* (MOUNT_UFS) or *dir* does not exist.

[ENAMETOOLONG] The pathname of *fspec* (MOUNT_UFS) or *dir* was too long.

[EACCES] Search permission is denied for a component of the path prefix of *fspec* (MOUNT_UFS) or *dir*.

[ELOOP] Too many symbolic links were encountered in translating the pathname of *fspec* (MOUNT_UFS) or *dir*.

[ENODEV] *type* is invalid, or the kernel does not support the requested type.

[ENOTBLK] *fspec* (MOUNT_UFS) is not a block device.

[ENXIO] The major device number of *fspec* (MOUNT_UFS) is out of range (this indicates no device driver exists for the associated hardware).

[EPFNOSUPPORT] The address family specified in *addr* (MOUNT_NFS) is not supported for NFS file systems. (Currently only AF_INET is supported).

[EINVAL] The *wsize, rsize, timeo, retrans* specified in the *nfs_args* structure was invalid. (MOUNT_NFS)

[EBUSY] *dir* is not a directory, or another process currently holds a reference to it.

[EBUSY] *fspec* (MOUNT_UFS) is already mounted.

[EBUSY] Insufficient space exists to hold the mount table entry or the cylinder group information for the file system.

[EBUSY] The super block for the file system contained bad data; i.e. it had a bad magic number or an out of range block size.

[EFAULT] One of the arguments passed to *mount* evaluates to an address outside the process's allocated address space.

[EIO] An I/O error occurred while reading from or writing to the file system.

SEE ALSO

unmount(2), mountd(8C), mount(8)

NAME

mremap – change attributes of a memory region in a process' address space

SYNOPSIS

```
#include    <sys/types.h>
#include    <sys/mman.h>

mremap(addr, len, prot, share)
caddr_t addr;
unsigned *len, prot, share;
```

DESCRIPTION

The *mremap* system call allows a user to change the attributes for a segment (for example: bss, stack, data) of a process' address space. This includes regions previously mapped with *mmap*(2) as well as arbitrary segments of a process' text, bss, tbss, data, tdata, stack, or tstack regions.

addr is the beginning virtual address of the memory region to which the new attributes apply. *addr* must be a multiple of NBPG.

len specifies the length of the segment. If *len* is not a multiple of NBPG then it is rounded up to a multiple of NBPG. When the MAP_GROWAUTO option is specified in the *share* field, then *len* is used to specify the limit for the size of the region and contains the current size of the segment being remapped.

The *prot* specifies the desired page protection bits in the page table entries and is composed of the logical-OR of PROT_READ, PROT_WRITE, and PROT_EXEC. The PROT_READ, PROT_WRITE, and PROT_EXEC, correspond to the ability to read, write, or execute the remapped pages.

The *share* field is the logical or of the attributes desired for the memory segments within the range specified by *addr* and *len*. Valid options are MAP_SHARED, MAP_PRIVATE, MAP_GROWAUTO, MAP_GROWDOWN, MAP_GROWUP, and MAP_INHERIT. When the MAP_GROWAUTO Option is specified, only the segment that contains the address *addr* is remapped.

RETURN VALUE & ERRORS

mremap returns 0 if the remapping was successful. If unsuccessful, mremap returns a -1 with errno set as follows:

[EINVAL] *addr* points to an invalid address, attempted to remap a MAP_SHARED segment as MAP_PRIVATE, or share contains an option set not contained in the above list of valid options.

[EACCES] The specified protection conflicts with the permissions granted by the file corresponding to that mapped segment.

[ENOMEM] There is insufficient swap space or memory to grant the request.

SEE ALSO

mmap(2), munmap(2), msleep(2), mwakeup(2), msync(2), tas(3)
Using Shared Memory

NAME

msleep, mwakeup – shared memory synchronization primitives

SYNOPSIS

```
#include    <sys/types.h>
msleep(sem)
semaphore *sem;
mwakeup(sem)
semaphore *sem;
```

DESCRIPTION

These routines provide services analogous to the kernel sleep and wakeup functions interpreted on the domain of a shared file.

sem is a pointer to a machine dependent semaphore structure, defined in *sys/types.h*. On the CONVEX Computer, the structure is:

```
struct      semaphore   {     /* machine dependent */
      char  lock;       /* the tas lock */
      char  awakened;   /* waiters have been awakened */
      };
typedef struct semaphore semaphore;
```

sem must be in a shared memory region, with at least **PROT_READ** and **PROT_WRITE**. The **MAP_HASSEMAPHORE** flag must have been set when the *mmap* call was executed. The region may be mapped in any type– MAP_FILE, MAP_ANON, or MAP_DEVICE.

msleep arranges that the calling process relinquish the processor if the semaphore *lock* is set. If *lock* is not set, *msleep* returns immediately. The process will remain asleep until some other process issues an *mwakeup* on the same semaphore, or a signal is received. *mwakeup* awakens any processes waiting on the same semaphore. The *awakened* byte should not be modified by user programs; it is used by the kernel to know if someone is waiting on the semaphore for a wakeup.

User programs should use *mset* and *mclear* instead of *msleep* and *mwakeup* if possible, since the latter are less machine dependent.

RETURN VALUE & ERRORS

msleep and *mwakeup* return zero if successful. If unsuccessful, they return a -1 with errno set as follows:

[EFAULT] *sem* did not point to a valid address for the semaphore structure, or PROT_READ or PROT_WRITE were not set for the mapped region.

[EINVAL] *fd* was not mapped into any shared memory segment.

[EINTR] A signal was received which interrupted the system call.

SEE ALSO

mmap(2), mset(3), tas(3)

NAME

msync – synchronize shared memory segment with file system

SYNOPSIS

msync(addr,len)
caddr_t addr;
int len;

DESCRIPTION

The *msync* system call causes dirty pages in shared memory segments (mapped MAP_FILE) to be flushed back to the buffer cache, so that subsequent *sync* and *fsync* calls will process the correct data for the file. All the pages from *addr* to *addr+len* are flushed.

RETURN VALUE & ERRORS

msync returns zero if successful. If unsuccessful, *msync* returns a -1 with errno set as follows:

[EFAULT] The region defined by *addr* and *len* includes pages invalid (unmapped) pages.

SEE ALSO

mmap(2)

NAME

munmap – unmap memory

SYNOPSIS

```
#include      <sys/types.h>
#include      <sys/mman.h>
munmap(addr,size)
caddr_t addr;
```

DESCRIPTION

The munmap system call unmaps *size* bytes of memory beginning at *addr*. Subsequent references to virtual addresses in the unmapped area will generate bus errors (SIGBUS).

The actual range is *addr*, rounded down to a page boundary, plus *size*, rounded up to a page length.

If *size* is 0, and *addr* is the beginning of a memory segment, the entire segment is unmapped.

The unmap range may be an entire memory segment, a partial memory segment, or multiple memory segments. Unmapped memory within the range is ignored, but *addr* must begin within a memory segment.

All memory, except thread memory, is eligible for unmapping. This includes shared memory, text, stack, data, and bss.

RETURN VALUE & ERRORS

munmap returns zero if successful. If unsuccessful, munmap returns a -1 with errno set as follows:

[EINVAL] *addr* does not begin within a memory segment.

SEE ALSO

mmap(2)

NAME

nfssvc, async_daemon – NFS daemons

SYNOPSIS

nfssvc(sock)
int sock;

async_daemon()

DESCRIPTION

nfssvc starts an NFS daemon listening on socket *sock*. The socket must be AF_INET, and SOCK_DGRAM (protocol UDP/IP). The system call will return only if the process is killed.

async_daemon implements the NFS daemon that handles asynchronous I/O for an NFS client. The system call never returns.

BUGS

These two system calls allow kernel processes to have user context.

SEE ALSO

mountd(8C)

NAME

open – open a file for reading or writing, or create a new file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(path, flags, ...)
char *path;
int  flags;
```

DESCRIPTION

open() opens the file *path* for reading and/or writing, as specified by the *flags* argument and returns a descriptor for that file. The *flags* argument may indicate the file is to be created if it does not already exist (by specifying the O_CREAT flag), in which case the file is created with a mode specified as a third argument, *mode*, of type **mode_t**. In this case, *mode* is described as in *chmod*(2) and modified by the process's umask value (see *umask*(2)).

path is the address of a NULL terminated string of ASCII characters representing a path name. *flags* values are constructed by *OR*ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_CREAT If the file exists, this flag has no effect. Otherwise, the owner ID of the file is set to the effective user ID of the process, the group ID of the file is set to the group ID of the directory in which the file is created, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows (see *creat*(2)):

All bits set in the file mode creation mask of the process are cleared. See *umask*(2).

The "save text image after execution" bit of the mode is cleared. See *chmod*(2).

O_EXCL If O_EXCL and O_CREAT are set, *open()* will fail if the file exists. This can be used to implement a simple exclusive access locking mechanism. If O_EXCL is set and the last component of the pathname is a symbolic link, the open will fail even if the symbolic link points to a non-existent name.

O_NOCTTY If set, and *path* indicates a terminal device, the terminal device will not become the controlling terminal for the process.

O_NONBLOCK

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NONBLOCK is set:

An *open()* for reading-only will return without delay. An *open()* for writing-only will return an error if no process currently has the file open for reading.

If O_NONBLOCK is clear:

An *open()* for reading-only will block until a process opens the file for writing. An *open()* for writing-only will block

until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NONBLOCK is set:

The open will return without waiting for carrier. The first time the process attempts to perform I/O on the open file it will block (not currently implemented).

If O_NONBLOCK is clear:

The open will block until carrier is present.

When opening a file on a filesystem under the control of a migration daemon:

If O_NONBLOCK is set:

The process wants to receive an *ENOSPC* error instead of blocking.

If O_NONBLOCK is clear:

The process will block until sufficient blocks or fragments become available to satisfy the request.

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new descriptor is set to remain open across *execve*(2) system calls; see *close*(2) and *fcntl*(2).

There is a system enforced limit on the number of open file descriptors per process with a value returned by the *getdtablesize*(2) call.

RETURN VALUE

The value –1 is returned if an error occurs, and external variable *errno* is set to indicate the cause of the error. Otherwise a non-negative numbered file descriptor for the new open file is returned.

ERRORS

open() fails if:

[ENOTDIR] A component of the path prefix of *path* is not a directory.

[ENOTDIR] *path* ends in a slash, but is not a directory.

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG]
The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.

[ENOENT] O_CREAT is not set and the named file does not exist.

[ENOENT] A component of the path prefix of *path* does not exist.

[ENOENT] The *path* argument points to an empty string.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[EACCES] Search permission is denied for a component of the path prefix of *path*.

[EACCES] The required permissions (for reading and/or writing) are denied for the file named by *path*.

[EACCES] The file referred to by *path* does not exist, O_CREAT is specified, and the directory in which it is to be created does not permit writing.

[EINTR] The *open()* call was interrupted by a signal.

[EISDIR] The named file is a directory, and the arguments specify it is to be opened for writing.

[ENXIO] O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.

[ENXIO] The file is a character special or block special file, and the associated device does not exist.

[EMFILE] The system limit for open file descriptors per process has already been reached.

[ENFILE] The system file table is full.

[ENOSPC] The file does not exist, O_CREAT is specified, and the directory in which the entry for the new file is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENOSPC] The file does not exist, O_CREAT is specified, and there are no free inodes on the file system on which the file is being created.

[EDQUOT] The file does not exist, O_CREAT is specified, and the directory in which the entry for the new file is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EDQUOT] The file does not exist, O_CREAT is specified, and the user's quota of inodes on the file system on which the file is being created has been exhausted.

[EROFS] The named file does not exist, O_CREAT is specified, and the file system on which it is to be created is a read-only file system.

[EROFS] The named file resides on a read-only file system, and the file is to be opened for writing.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed and the *open()* call requests write access.

[EIO] An I/O error occurred while reading from or writing to the file system.

[EIO] The named file corresponds to a tape device which is offline, or one which was opened for write access without having a write enable ring present on the tape.

[EFAULT] *path* points outside the process's allocated address space.

[EEXIST] O_EXCL and O_CREAT were both specified and the file exists.

[EOPNOTSUPP] An attempt was made to open a socket (not currently implemented).

BACKWARD COMPATIBILITY

In previous versions of the operating system, the null string for *path* was a synonym for ., the current directory.

The O_NONBLOCK flag was previous known as O_NDELAY. O_NDELAY had slightly different semantics, in that it could modify other references to the same file which did not share the same file descriptor; in that sense, it was more like *ioctl()*.

SEE ALSO

chmod(2), close(2), dup(2), fcntl(2), lseek(2), read(2), write(2), umask(2)

NAME

pattach – process attach

SYNOPSIS

#include <sys/file.h>

pattach (pid, flags)
int pid, flags;

DESCRIPTION

pattach creates a connection to a process and returns a descriptor that may be used to read, write, or control the execution of another process.

The *pid* parameter specifies the process id that should be attached. The *flags* parameter is one of the following values

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_EXCL	open for exclusive execution access

Opening a process with O_EXCL grants the caller exclusive control of the execution of the process. A process may be opened only if it has the same effective user ID as the caller and is in the same process group. A superuser or process in group *daemon* may open any other process.

RETURN VALUE

If successful, a descriptor referencing the process is returned. If unsuccessful, a –1 is returned, and the global variable *errno* is set to indicate the error.

ERRORS

The *pattach* call fails if:

[EACCES]	The process is already opened for exclusive access.
[EPERM]	The caller is not the owner of the process.
[ESRCH]	The process does not exist.
[EMFILE]	The open file limit has been reached.
[EINVAL]	An invalid *flags* parameter was specified.

SEE ALSO

pi(4)

NAME

pgetregid – get a process' real and effective group ID

SYNOPSIS

pgetregid(pid, rgid, egid)
int pid;
short *rgid, *egid;

DESCRIPTION

The real and effective group ID's of the process whose ID is *pid* are returned in the variables pointed to by the arguments *rgid* and *egid*.

Either *rgid* or *egid* may be a null pointer, in which case the respective ID is not copied.

RETURN VALUE

Upon successful completion, a value of **0** is returned. Otherwise, a value of **–1** is returned and *errno* is set to indicate the error.

ERRORS

[ESRCH] The process specified does not exist.

[EFAULT] The *addr* parameter is not in a writable part of the user address space.

SEE ALSO

getgid(2), psetregid(2), setreuid(2), setuid(3),
"Accounting" chapter in the *CONVEX System Manager's Guide*.

NAME

pipe – create an interprocess communication channel

SYNOPSIS

```
#include <limits.h>
#include <unistd.h>

int pipe(fildes)
int fildes[2];
```

DESCRIPTION

The *pipe()* system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor *fildes*[1] up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor *fildes*[0] will pick up the data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent *fork(2)* calls) will pass data through the pipe with *read(2)* and *write(2)* calls.

The shell has a syntax to set up a linear array of processes connected by pipes.

read(2) on an empty pipe (no buffered data) with only one end (all write file descriptors closed) returns an end-of-file.

A signal is generated if a write on a pipe with only one end is attempted.

The O_NONBLOCK flag is clear for both file descriptors created by *pipe()*; the *fcntl()* call may be used to set the O_NONBLOCK flag.

RETURN VALUE

The function value zero is returned if the pipe was created; –1 if an error occurred.

ERRORS

The *pipe()* call will fail if:

[EMFILE]	Too many descriptors are active.
[EFAULT]	The *fildes* buffer is in an invalid area of the process's address space.

SEE ALSO

sh(1), read(2), write(2), fork(2), socketpair(2)

NOTES

Should more than PIPE_BUF bytes be necessary in any pipe among a loop of processes, deadlock will occur.

Pipes are really a special case of the *socketpair*(2) call and, in fact, are implemented as such in the system.

NAME

profil, lprofil – execution time profile

SYNOPSIS

profil(buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;

lprofil(lbuff, bufsiz, offset, scale)
char *lbuff;
int bufsiz, offset, scale;

DESCRIPTION

Buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick (10 milliseconds); *offset* is subtracted from it, and the result multiplied by *scale*. If the resulting number corresponds to an unsigned short inside *buff*, that unsigned short is incremented. In the case of *lprofil*, the buffer indicated by *lbuff* is maintained as a buffer of unsigned longs.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0x10000 gives a 1-1 mapping of instruction half-words to locations in *buff*; 0x8000 maps each pair of instruction half-words together. 0x2 maps 64K-byte regions of instructions onto locations of buff.

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *execve* is executed, but remains on in child and parent both after a *fork*. Profiling is turned off if an update in *buff* would cause a memory fault.

RETURN VALUE

A 0, indicating success, is always returned.

SEE ALSO

prof(1)(optional product), gprof(1)(optional product), getitimer(2), monitor(3)

BUGS

Programs using the *lprofil* system call will frequently produce the diagnostic "Bad system call: core dumped" when run on a version of the operating system older than 7.1.

NAME

psetregid – set a process' real and effective group ID

SYNOPSIS

psetregid(pid, rgid, egid)
int pid, rgid, egid;

DESCRIPTION

The real and effective group ID's of the process whose ID is *pid* are set to the arguments. Only the superuser may use this system call.

If a value of **–1** is supplied for either the real or effective group ID, the system does not change the respective ID.

RETURN VALUE

Upon successful completion, a value of **0** is returned. Otherwise, a value of **–1** is returned and *errno* is set to indicate the error.

ERRORS

[EPERM] The current process is not the superuser.

[ESRCH] The process specified does not exist.

SEE ALSO

getgid(2), pgetregid(2), setreuid(2), setuid(3),
"Accounting" chapter in the *CONVEX System Manager's Guide.*

NAME

quotactl – manipulate disk quotas

SYNOPSIS

```
#include <sys/types.h>
#include <ufs/quota.h>

quotactl(cmd, special, uid, addr)
int cmd;
char *special;
int uid;
caddr_t addr;
```

DESCRIPTION

The *quotactl* call manipulates disk quotas. The *cmd* parameter indicates a command to be applied to the user ID *uid*. *special* is a pointer to a null-terminated string containing the path name of the block special device for the file system being manipulated. The block special device must be mounted. *addr* is the address of an optional, command specific, data structure which is copied in or out of the system. The interpretation of *addr* is given with each command below.

Q_QUOTAON

Turn on quotas for a file system. *addr* is a pointer to a null terminated string containing the path name of file containing the quotas for the file system. The quota file must exist; it is normally created with the *quotacheck*(8) program. This call is restricted to the super-user.

Q_QUOTAOFF

Turn off quotas for a file system. This call is restricted to the super-user.

Q_GETQUOTA

Get disk quota limits and current usage for user *uid*. *addr* is a pointer to a struct dqblk structure (defined in *<ufs/quota.h>*). Only the super-user may get the quotas of a user other than himself. If *uid* is 0, then the timelimit fields of the returned dqblk structure are the quota time limits for the filesystem, *special*. Otherwise, the timelimit fields are the system times at which the user's hardlimits for filesystem, *special*, will be reached.

Q_SETQUOTA

Set disk quota limits and current usage for user *uid*. *addr* is a pointer to a struct dqblk structure (defined in *<ufs/quota.h>*). This call is restricted to the superuser. If *uid* is 0, then the quota time limits for filesystem, *special*, are set to the time limit fields in the dqblk structure.

Q_SETQLIM

Set disk quota limits for user *uid*. *addr* is a pointer to a struct dqblk structure (defined in *<ufs/quota.h>*). This call is restricted to the super-user. If *uid* is 0, then the quota time limits for filesystem, *special*, are set to the time limit fields in the dqblk structure.

Q_SYNC

Update the on-disk copy of quota usages for the *special* filesystem. This call is restricted to the super-user. If *special* is 0, then quotas for all filesystems with quotas enabled will effectively be *sync*ed.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

A *quotactl* call will fail when one of the following occurs:

[EINVAL] *cmd* is invalid.

[EPERM] The call is privileged and the caller was not the super-user.

[EINVAL] The *special* is not a mounted file system or is a mounted file system without quotas enabled.

[ENOTBLK] The *special* parameter is not a block device.

[EFAULT] An invalid *addr* is supplied; the associated structure could not be copied in or out of the kernel.

[EACCES] The *addr* parameter is being interpreted as the path of a quota file which exists but is either not a regular file or is not on the file system pointed to by the *special* parameter.

[EINVAL] An internal error (such as an I/O error) has occurred in the quota system.

[ENOTDIR] A pathname lookup was being performed on either *special* or *addr* (Q_QUOTAON) and a component of the prefix was not a valid directory.

[ENOENT] A pathname lookup was being performed on either *special* or *addr* (Q_QUOTAON) and no such entry was found in the directory.

[EFAULT] An invalid *addr* is supplied; the associated structure could not be copied in or out of the kernel.

[EUSERS] The quota table is full.

SEE ALSO

quotaon(8), quotacheck(8)

BUGS

There should be some way to integrate this call with the resource limit interface provided by *setrlimit*(2) and *getrlimit*(2). Incompatible with Melbourne quotas.

NAME

read – read from a file

SYNOPSIS

cc = read(d, buf, nbytes)
int cc, d;
char *buf;
unsigned nbytes;

DESCRIPTION

read() attempts to read *nbytes* of data from the object referenced by the descriptor *d* into the buffer pointed to by *buf*.

On objects capable of seeking, the *read()* starts at a position given by the pointer associated with *d*. (See *lseek*(2)). Upon return from *read()*, the pointer is incremented by the number of bytes actually read.

Objects that are not capable of seeking always read from the current position. The value of the pointer associated with such an object is undefined.

Note: For read access to a directory, use either the *readdir*(3) or *getdirentries*(2) functions. (Directory access via *read()* is no longer available.)

The system guarantees to read the number of bytes requested only if the descriptor references a file that has that many bytes left before the end-of-file.

When attempting to read from a descriptor associated with an empty pipe, socket, or FIFO:

If O_NONBLOCK is set, the read will return a –1 and *errno* will be set to EAGAIN.

If O_NONBLOCK is clear, the read will block until data is written to the pipe or the file is no longer open for writing.

When attempting to read from a descriptor associated with a tty that has no data currently available:

If O_NONBLOCK is set, the read will return a –1 and *errno* will be set to EAGAIN.

If O_NONBLOCK is clear, the read will block until data becomes available.

If O_NONBLOCK is set and less data is available than requested by the *read()*, only the data that is available is returned, and the count indicates how many bytes of data were actually read.

RETURN VALUE

If successful, the number of bytes actually read is returned unless the read was asynchronous (see EXTENSIONS below), in which case the number of bytes requested is returned. If unsuccessful, a –1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

read() will fail if one or more of the following are true:

[EAGAIN]	The file was marked for non-blocking I/O (O_NONBLOCK flag set), and no data were ready to be read.
[EBADF]	*d* is not a valid file descriptor open for reading.
[EINVAL]	The requested transfer size is too big for a single request. In order to prevent possible memory deadlocks, the kernel will refuse to do transfers larger than approximately half the size of physical memory.
[EINTR]	A *read()* from a slow device was interrupted before any data arrived by the delivery of a signal.
[EISDIR]	*d* refers to a directory that is on a file system mounted using the NFS.
[EFAULT]	*buf* points outside the allocated address space.

[EIO] The process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group of the process orphaned. (This would most commonly result from the user logging out with a program left running in the background, and this program attempts to read from standard input).

[EIO] An I/O error occurred while reading from or writing to the file system.

[ENODMON] *Migration only:* The file referenced by *d* is under daemon control, but no daemon is present.

EISMIGRATED *Migration only:* A block in the file reference by *d* is migrated, and the migration daemon was unable to stage the file in from secondary storage.

CONVEX EXTENSIONS

read() may operate synchronously (default) or asynchronously, depending on the FASIO flag set with the *fcntl(2)* system call. Synchronous reads suspend the caller until the system has processed the read request and return the number of bytes actually read and placed in the buffer or 0 if end-of-file has been reached. The file position pointer is incremented by the number of bytes actually read. Asynchronous reads return before the request has been fully processed and, unless there are errors in the arguments passed, always return the number of bytes requested whether it is actually possible to complete the request. The file position pointer is incremented by the number of bytes requested. To determine what actually happened during asynchronous transfers, use *asiostat*(2).

BACKWARD COMPATIBILITY

The O_NONBLOCK mode of operation was formerly known as O_NDELAY.

The current implementation of O_NONBLOCK is slightly different than the old O_NDELAY; O_NONBLOCK will affect no processes other than the caller, whereas O_NDELAY would affect all processes with file descriptors open for that device.

The error return EWOULDBLOCK occurs in backward compatibility mode in those situations that now return EAGAIN.

SEE ALSO

asiostat(2), dup(2), fcntl(2), open(2), pipe(2), readv(2), select(2), socket(2), socketpair(2)

NAME

readlink – read value of a symbolic link

SYNOPSIS

cc = readlink(path, buf, bufsiz)
int cc;
char *path, *buf;
int bufsiz;

DESCRIPTION

Readlink places the contents of the symbolic link *name* in the buffer *buf* which has size *bufsiz*. The contents of the link are not null terminated when returned.

RETURN VALUE

The call returns the count of characters placed in the buffer if it succeeds, or a –1 if an error occurs, placing the error code in the global variable *errno*.

ERRORS

Readlink will fail and the file mode will be unchanged if:

[EINVAL]	The *path* argument contained a byte with the high-order bit set.
[ENOENT]	The pathname was too long.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[ENXIO]	The named file is not a symbolic link.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not the super-user.
[EINVAL]	The named file is not a symbolic link.
[EFAULT]	*Buf* extends outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

SEE ALSO

stat(2), symlink(2)

NAME

readv – read scattered data

SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>

cc = readv(d, iov, iovcnt)
int cc, d;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

readv() attempts to read from the object referenced by the descriptor *d*, scattering the input data into the *iovcnt* buffers specified by the members of the *iovec* array: iov[0], iov[1], ..., iov[iovcnt – 1].

The *iovec* structure is defined as

```
struct iovec {
        caddr_t iov_base;
        int     iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. *readv()* will always fill an area completely before proceeding to the next.

On objects capable of seeking, the *readv()* starts at a position given by the pointer associated with *d*. See *lseek*(2). Upon returning from *readv()*, the pointer is incremented by the number of bytes actually read.

Objects that are not capable of seeking always read from the current position. The value of the pointer associated with such an object is undefined.

Note: For read access to a directory, use *readdir*(3) or *getdirentries*(2). function. (Directory access using *readv()* is no longer supported.)

readv() may operate synchronously (default) or asynchronously, depending on the FASIO flag set with the *fcntl*(2) system call. Synchronous reads suspend the caller until the system has processed the read request and return the number of bytes actually read and placed in the buffer or 0 if end-of-file has been reached. The file position pointer is incremented by the number of bytes actually read. Asynchronous reads return before the request has been fully processed and unless there are errors in the arguments passed, always return the number of bytes requested whether it is actually possible to complete the request. The file position pointer is incremented by the number of bytes requested. To determine what actually happened during asynchronous transfers, use *asiostat*(2).

The system guarantees to read the number of bytes requested only if the descriptor references a file that has that many bytes left before the end-of-file.

When attempting to read from a descriptor associated with an empty pipe, socket, or FIFO:

If O_NONBLOCK is set, the read will return a –1 and *errno* will be set to EWOULDBLOCK.

If O_NONBLOCK is clear, the read will block until data is written to the pipe or the file is no longer open for writing.

When attempting to read from a descriptor associated with a tty that has no data currently available:

If O_NONBLOCK is set, the read will return a –1 and *errno* will be set to EWOULDBLOCK.

If O_NONBLOCK is clear, the read will block until data becomes available.

If O_NONBLOCK is set and less data are available than are requested by the *read()* or *readv()*, only the data that are available are returned, and the count indicates how many bytes of data were actually read.

RETURN VALUE

If successful, the number of bytes actually read is returned, unless the read was asynchronous, in which case the number of bytes requested is returned. If unsuccessful, a –1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

readv() will fail if one or more of the following are true:

[EINVAL]	The requested transfer size is too big for a single request. In order to prevent possible memory deadlocks, the kernel will refuse to do transfers larger than approximately half the size of physical memory.
[EBADF]	*d* is not a valid file descriptor open for reading.
[EISDIR]	*d* refers to a directory that is on a file system mounted using the NFS.
[EFAULT]	*buf* points outside the allocated address space.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EINTR]	A read from a slow device was interrupted before any data arrived by the delivery of a signal.
[EWOULDBLOCK]	The file was marked for non-blocking I/O, and no data were ready to be read.
[EINVAL]	*iovcnt* was either less than or equal to 0 or greater than 16.
[EINVAL]	For asynchronous requests, a maximum of one *iov* region may point to a given logical page.
[EINVAL]	One of the *iov_len* values in the *iov* array was negative.
[EINVAL]	The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.
[EFAULT]	Part of *iov* points outside the process's allocated address space.
[EINVAL]	Two or more *iov* buffers share the same virtual memory page, and the mode is asynchronous. (This restriction is necessary because the pages are locked in memory during asynchronous transfers.)
[EIO]	A read from a magnetic tape encountered an end of tape marker.
[ENODMON]	*Migration only:* The file referenced by *d* is under daemon control, but no daemon is present.
EISMIGRATED	*Migration only:* A block in the file reference by *d* is migrated, and the migration daemon was unable to stage the file in from secondary storage.

SEE ALSO

asiostat(2), dup(2), fcntl(2), open(2), pipe(2), read(2), select(2), socket(2), socketpair(2)

NAME

reboot – reboot system or halt processor

SYNOPSIS

#include <sys/reboot.h>

reboot(howto)
int howto;

DESCRIPTION

Reboot reboots the system, and is invoked automatically in the event of unrecoverable system failures. *Howto* is a mask of options passed to the bootstrap program. The system call interface permits only RB_HALT or RB_AUTOBOOT to be passed to the reboot program; the other flags are used in scripts stored on the console storage media, or used in manual bootstrap procedures. When none of these options (e.g. RB_AUTOBOOT) is given, the system is rebooted from file "vmunix". An automatic consistency check of the disks is then normally performed.

The bits of *howto* are:

RB_HALT
: the processor is simply halted; no reboot takes place. RB_HALT should be used with caution.

RB_ASKNAME
: Interpreted by the bootstrap program itself, causing it to inquire as to what file should be booted. Normally, the system is booted from the file "vmunix" without asking.

RB_SINGLE
: Normally, the reboot procedure involves an automatic disk consistency check and then multi-user operations. RB_SINGLE prevents the consistency check, rather simply booting the system with a single-user shell on the console. RB_SINGLE is interpreted by the *init*(8) program in the newly booted system. This switch is not available from the system call interface.

Only the super-user may *reboot* a machine.

RETURN VALUES

If successful, this call never returns. Otherwise, a –1 is returned and an error is returned in the global variable *errno*.

ERRORS

[EPERM] The caller is not the super-user.

SEE ALSO

halt(8), init(8)

NAME

recv, recvfrom, recvmsg – receive a message from a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

cc = recv(s, buf, len, flags)
int cc, s;
char *buf;
int len, flags;

cc = recvfrom(s, buf, len, flags, from, fromlen)
int cc, s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;

cc = recvmsg(s, msg, flags)
int cc, s;
struct msghdr msg[];
int flags;
```

DESCRIPTION

Recv, *recvfrom*, and *recvmsg* are used to receive messages from a socket.

The *recv* call is normally used only on a *connected* socket (see *connect*(2)), while *recvfrom* and *recvmsg* may be used to receive data on a socket whether it is in a connected state or not.

If *from* is non-zero, the source address of the message is filled in. *Fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned in *cc*. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see *socket*(2)).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see *ioctl*(2)) in which case a *cc* of –1 is returned with the external variable errno set to EWOULDBLOCK.

The *select*(2) call may be used to determine when more data arrives.

The *flags* argument to a recv call is formed by *or*'ing one or more of the values,

```
#define MSG_OOB         0x1     /* process out-of-band data */
#define MSG_PEEK        0x2     /* peek at incoming message */
```

The *recvmsg* call uses a *msghdr* structure to minimize the number of directly supplied parameters. This structure has the following form, as defined in *<sys/socket.h>*:

```
struct msghdr {
        caddr_t msg_name;               /* optional address */
        int     msg_namelen;            /* size of address */
        struct  iovec *msg_iov;         /* scatter/gather array */
        int     msg_iovlen;             /* # elements in msg_iov */
        caddr_t msg_accrights;          /* access rights sent/received */
        int     msg_accrightslen;
};
```

Here *msg_name* and *msg_namelen* specify the destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter gather locations, as described in *read*(2). A buffer to receive any

access rights sent along with the message is specified in *msg_accrights*, which has length *msg_accrightslen*. Access rights are currently limited to file descriptors, which each occupy the size of an **int**.

RETURN VALUE

These calls return the number of bytes received, or –1 if an error occurred.

ERRORS

The calls fail if:

[EBADF]	The argument *s* is an invalid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking and the receive operation would block.
[EINTR]	The receive was interrupted by delivery of a signal before any data was available for the receive.
[EFAULT]	The data was specified to be received into a non-existent or protected part of the process address space.
[ENOTCONN]	No *connect* had been done on a socket whose protocol requires connections.

SEE ALSO

fcntl(2), read(2), send(2), select(2), getsockopt(2), socket(2)

NAME

rename – change the name of a file

SYNOPSIS

```
#include <unistd.h> /* POSIX */
#include <stdio.h>  /* ANSI C */

int rename(const char *from, const char *to);
```

DESCRIPTION

rename() causes the link named *from* to be renamed as *to*. If *to* exists, then it is removed first. Both *from* and *to* must be of the same type (that is, both directories or both non-directories) and must reside on the same file system.

rename() guarantees that an instance of *to* will always exist even if the system should crash in the middle of the operation.

RETURN VALUE

A **0** value is returned if the operation succeeds, otherwise *rename* returns a nonzero value and the global variable *errno* indicates the reason for the failure.

ERRORS

rename() will fail and neither of the argument files will be affected if any of the following are true:

[ENOTEMPTY] The destination directory is not empty or has links to it.

[EISDIR] The destination is a directory and the source is not a directory.

[ENOTDIR] The destination is not a directory and the source is a directory.

[ENOTDIR] A component of either path prefix is not a directory.

[ENOENT] A component of either path prefix does not exist.

[EACCES] A component of either path prefix denies search permission.

[ENOENT] The file named by *from* does not exist.

[ENOENT] *from* or *to* points to an empty string.

[EPERM] The file named by *from* is a directory, and the effective user ID is not superuser.

[EXDEV] The link named by *to* and the file named by *from* are on different logical devices (file systems). Note that this error code will not be returned if the implementation permits cross-device links.

[EACCES] The requested link requires writing in a directory with a mode that denies write permission.

[EROFS] The requested link requires writing in a directory on a read-only file system.

[EFAULT] *from* or *to* points outside the process' allocated address space.

[EINVAL] *from* is a parent directory of *to*.

[ESTATECHANGED] *MIGRATION ONLY:* The target file changed state during a call to a migration daemon. There are a number of cases to consider. If *to* existed before the call to the daemon, then it no longer exists. If the target file *to* did not exist prior to the call to the daemon, it now exists. If *from* was removed, then the operation may still have suceeded; *from* is now renamed to *to*, but the operating system was unable to remove the old

from as part of the rename operation.

BACKWARD COMPATIBILITY

Previous versions of the operating system interpreted the empty path name as a synonym for the current directory.

Previous versions of the operating system had slightly different values of errno:

[EISDIR] The destination is a file, but the source is a directory.

[ENOTDIR] A component of either path prefix is not a directory.

NOTE

The system can deadlock if a loop in the file system graph is present. This loop takes the form of an entry in directory **a** (e.g., **a/foo**) being a hard link to directory **b**, and an entry in directory **b** (e.g., **b/foo**) being a hard link to directory **a**. When such a loop exists and two separate processes attempt to perform **rename a/foo b/bar** and **rename b/bar a/foo**, respectively, the system may deadlock attempting to lock both directories for modification. Hard links to directories should be replaced by symbolic links by the system administrator. (Note that only root may create a hard link to a directory in the first place.)

SEE ALSO

open(2), link(2)

NAME

rmdir – remove a directory file

SYNOPSIS

```
#include <unistd.h>

rmdir(path)
char *path;
```

DESCRIPTION

rmdir() removes a directory file whose name is given by *path*. The directory must not have any entries other than "." and "..".

RETURN VALUE

A 0 is returned if the remove succeeds; otherwise a –1 is returned and an error code is stored in the global location *errno*.

ERRORS

The named file is removed unless one or more of the following are true:

[ENOTEMPTY] The named directory contains files other than "." and ".." in it.

[EINVAL] The pathname contains a character with the high-order bit set.

[ENAMETOOLONG] The pathname was too long.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist, or *path* points to an empty string.

[EACCES] A component of the path prefix denies search permission.

[EACCES] Write permission is denied on the directory containing the link to be removed.

[EBUSY] The directory to be removed is the mount point for a mounted file system.

[EROFS] The directory entry to be removed resides on a read-only file system.

[EFAULT] *path* points outside the process's allocated address space.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

BACKWARD COMPATIBILITY

Previous versions of the operating system interpreted the empty path name as a synonym for the current directory.

SEE ALSO

mkdir(2), unlink(2)

NAME

select – synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/types.h>
#include <sys/time.h>

nfound = select(nfds, readfds, writefds, exceptfds, timeout)
int nfound, nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;

FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ISSET(fd, &fdset)
FD_ZERO(&fdset)
int fd;
fd_set fdset;
```

DESCRIPTION

select examines the I/O descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first *nfds* descriptors are checked in each set; i.e. the descriptors from 0 through *nfds*-1 in the descriptor sets are examined. On return, *select* replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned in *nfound*.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: *FD_ZERO(&fdset)* initializes a descriptor set *fdset* to the null set. *FD_SET(fd, &fdset)* includes a particular descriptor *fd* in *fdset*. *FD_CLR(fd, &fdset)* removes *fd* from *fdset*. *FD_ISSET(fd, &fdset)* is nonzero if *fd* is a member of *fdset*, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to *FD_SETSIZE*, which is normally at least equal to the maximum number of descriptors supported by the system.

If *timeout* is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, the select blocks indefinitely. To effect a poll, the *timeout* argument should be non-zero, pointing to a zero-valued timeval structure.

Any of *readfds*, *writefds*, and *exceptfds* may be given as zero pointers if no descriptors are of interest.

RETURN VALUE

select returns the number of ready descriptors that are contained in the descriptor sets, or –1 if an error occurred. If the time limit expires then *select* returns 0. If *select* returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified.

ERRORS

An error return from *select* indicates:

[EBADF]	One of the descriptor sets specified an invalid descriptor.
[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
[EINVAL]	The specified time limit is invalid. One of its components is negative or too large.

SEE ALSO

accept(2), connect(2), read(2), write(2), recv(2), send(2), getdtablesize(2)

BUGS

Although the provision of *getdtablesize*(2) was intended to allow user programs to be written independent of the kernel limit on the number of open files, the dimension of a sufficiently large bit field for select remains a problem. The default size FD_SETSIZE (256) is equal to the current kernel limit to the number of open files. However, in order to accommodate programs which might potentially use a larger number of open files with select, it is possible to increase this size within a program by providing a larger definition of FD_SETSIZE before the inclusion of <sys/types.h>.

select should probably return the time remaining from the original timeout, if any, by modifying the time value in place. This may be implemented in future versions of the system. Thus, it is unwise to assume that the timeout value will be unmodified by the *select* call.

NAME

send, sendto, sendmsg – send a message to a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

cc = send(s, msg, len, flags)
int cc, s;
char *msg;
int len, flags;

cc = sendto(s, msg, len, flags, to, tolen)
int cc, s;
char *msg;
int len, flags;
struct sockaddr *to;
int tolen;

cc = sendmsg(s, msg, flags)
int cc, s;
struct msghdr msg[];
int flags;
```

DESCRIPTION

Send, *sendto*, and *sendmsg* are used to transmit a message to another socket. *Send* may be used only when the socket is in a *connected* state, while *sendto* and *sendmsg* may be used at any time.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, then the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a *send*. Return values of –1 indicate some locally detected errors.

If no messages space is available at the socket to hold the message to be transmitted, then *send* normally blocks, unless the socket has been placed in non-blocking I/O mode. The *select*(2) call may be used to determine when it is possible to send more data.

The *flags* parameter may include one or more of the following:

```
#define MSG_OOB          0x1    /* process out-of-band data */
#define MSG_DONTROUTE    0x4    /* bypass routing, use direct interface */
```

The flag MSG_OOB is used to send "out-of-band" data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support "out-of-band" data. MSG_DONTROUTE is usually used only by diagnostic or routing programs.

See *recv*(2) for a description of the *msghdr* structure.

RETURN VALUE

The call returns the number of characters sent, or –1 if an error occurred.

ERRORS

[EBADF]	An invalid descriptor was specified.
[ENOTSOCK]	The argument *s* is not a socket.
[EFAULT]	An invalid user space address was specified for a parameter.
[EMSGSIZE]	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.
[EWOULDBLOCK]	The socket is marked non-blocking and the requested operation would block.

[ENOBUFS] The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.

[ENOBUFS] The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

[ENOTCONN] No connection had been established when a *send* operation was tried.

SEE ALSO

fcntl(2), recv(2), select(2), getsockopt(2), socket(2), write(2)

NAME

setaid – set a process' activity ID

SYNOPSIS

setaid(pid, aid)
int pid, aid;

DESCRIPTION

The activity ID of the process whose ID is *pid* is set to *aid.*

An activity ID may be any 32-bit value except **–1**. Attempts to set an activity ID to **–1** are quietly ignored.

Only the superuser may use this system call.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of **–1** is returned and *errno* is set to indicate the error.

ERRORS

[EPERM] The current process is not the superuser.

[ESRCH] The process specified does not exist.

SEE ALSO

getaid(2),
"Accounting" chapter in the *CONVEX System Manager's Guide.*

NAME

setgroups – set group access list

SYNOPSIS

#include <sys/param.h>

setgroups(ngroups, gidset)
int ngroups, *gidset;

DESCRIPTION

Setgroups sets the group access list of the current user process according to the array *gidset*. The parameter *ngroups* indicates the number of entries in the array and must be no more than NGROUPS, as defined in *<sys/param.h>*.

Only the superuser may set new groups.

RETURN VALUE

A **0** value is returned on success, **–1** on error, with an error code stored in *errno*.

ERRORS

The *setgroups* call will fail if:

[EINVAL]	*Ngroups* is too large.
[EPERM]	The caller is not the superuser.
[EFAULT]	The address specified for *gidset* is outside the process address space.

SEE ALSO

getgroups(2), initgroups(3X)

NAME

setpgid – set process group

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int setpgid(pid, pgrp)
pid_t pid,pgrp;
```

DESCRIPTION

setpgid() is used to cause process *pid* to join an existing process group or create a new process group within the session of the calling process. If *pid* is zero, then the process ID of the calling process is used. If *pgrp* is zero, then the process ID of the calling process is used.

The process group ID of a session leader may not be changed.

RETURN VALUE

setpgid() returns zero when the operation was successful. If the request failed, –1 is returned and the global variable *errno* indicates the reason.

ERRORS

setpgid() will fail and the process group will not be altered if one of the following occur:

[EACCES]	*pid* matches the process ID of a child process of the calling process and the child process has successfully executed one of the *exec* functions.
[EINVAL]	The value of *pgrp* is less than zero.
[EPERM]	The process indicated by *pid* is a session leader.
[EPERM]	The value of *pid* matches a process of the calling process, but the child is not in the same session as the calling process.
[EPERM]	The value of *pgrp* does not match the process indicated by *pid* and there is no process with a process group ID that matches the value of *pgrp* in the same session as the calling process.
[ESRCH]	The value of *pid* does not match the process ID of the calling process or of a child process of the calling process.

BACKWARD COMPATIBILITY

This function was previously known as *setpgrp()*. The functionality of *setpgrp()* was essentially that of *setpgid()* without the restrictions on session membership.

SEE ALSO

getpgrp(2)

NAME

setpgrp – set process group

SYNOPSIS

setpgrp(pid, pgrp)
int pid, pgrp;

DESCRIPTION

Setpgrp sets the process group of the specified process *pid* to the specified *pgrp*. If *pid* is zero, then the call applies to the current process.

If the invoker is not the super-user, then the affected process must have the same effective user-id as the invoker or be a descendant of the invoking process.

RETURN VALUE

Setpgrp returns when the operation was successful. If the request failed, –1 is returned and the global variable *errno* indicates the reason.

ERRORS

Setpgrp will fail and the process group will not be altered if one of the following occur:

[ESRCH] The requested process does not exist.

[EPERM] The effective user ID of the requested process is different from that of the caller and the process is not a descendent of the calling process.

SEE ALSO

getpgrp(2)

NAME

setregid – set real and effective group ID

SYNOPSIS

setregid(rgid, egid)
int rgid, egid;

DESCRIPTION

The real and effective group ID's of the current process are set to the arguments. Only the superuser may change the real group ID of a process. Unprivileged users may change the effective group ID to the real group ID, but to no other.

Supplying a value of **–1** for either the real or effective group ID forces the system to substitute the current ID in place of the **–1** parameter.

RETURN VALUE

Upon successful completion, a value of **0** is returned. Otherwise, a value of **–1** is returned and *errno* is set to indicate the error.

ERRORS

[EPERM] The current process is not the superuser and a change other than changing the effective group-ID to the real group-ID was specified.

SEE ALSO

getgid(2), setreuid(2), setuid(3)

NAME

setreuid – set real and effective user ID's

SYNOPSIS

setreuid(ruid, euid)
int ruid, euid;

DESCRIPTION

The real and effective user ID's of the current process are set according to the arguments. If *ruid* or *euid* is –1, the current uid is filled in by the system. Only the super-user may modify the real uid of a process. Users other than the super-user may change the effective uid of a process only to the real uid.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

[EPERM] The current process is not the super-user and a change other than changing the effective user-id to the real user-id was specified.

SEE ALSO

getuid(2), setregid(2), setuid(3)

NAME

setsid – create session and set process group ID

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t setsid()
```

DESCRIPTION

If the calling process is not a process group leader, the *setsid()* function creates a new session. The calling process is the leader of this new session, is the process group leader of a new process group, and has no controlling terminal. The process group ID of the calling process is set equal to the process ID of the calling process. The calling process is the only process in the new process group and the only process in the new session.

RETURN VALUE

Upon success, *setsid()* returns the process group ID of the calling process, which is in fact its process ID. Upon failure, –1 is returned and *errno* indicates the cause for failure.

ERRORS

setsid() will fail if any one of the following occur:

[EPERM]	The calling process is already a process group leader.
[EPERM]	The process group ID of a process other than the calling process matches the process ID of the calling process.

SEE ALSO

intro(2), getpgrp(2), setpgid(2)

NAME

shutdown – shut down part of a full-duplex connection

SYNOPSIS

shutdown(s, how)
int s, how;

DESCRIPTION

The *shutdown* call causes all or part of a full-duplex connection on the socket associated with *s* to be shut down. If *how* is 0, then further receives will be disallowed. If *how* is 1, then further sends will be disallowed. If *how* is 2, then further sends and receives will be disallowed.

DIAGNOSTICS

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

The call succeeds unless:

[EBADF] *S* is not a valid descriptor.

[ENOTSOCK] *S* is a file, not a socket.

[ENOTCONN] The specified socket is not connected.

SEE ALSO

connect(2), socket(2)

NAME

sigaction – examine and change signal action

SYNOPSIS

```
#include <signal.h>

struct sigaction {
        void    (*sa_handler)();
        int     sa_mask;
        int     sa_flags;
};

int sigaction(sig, vec, ovec)
int sig;
struct sigaction *vec, *ovec;
```

DESCRIPTION

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a *handler* to which a signal is delivered or specify that a signal is to be *blocked* or *ignored*. A process may also specify that a default action is to be taken by the system when a signal occurs.

All signals have the same *priority*. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. A global *signal mask* defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It may be changed with a *sigprocmask*(2) call or when a signal is delivered to the process.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process, it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally, the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself.

When a signal is delivered to a process, a new signal mask is installed for the duration of the process's signal handler (or until a *sigblock()* or *sigsetmask()* call is made). This mask is formed by taking the current signal mask, adding the signal to be delivered, and *or*ing in the signal mask associated with the handler to be invoked.

sigaction() assigns a handler for a specific signal. If *vec* is non-zero, it specifies a handler routine and mask to be used when delivering the specified signal. If *ovec* is non-zero, the previous handling information for the signal is returned to the user.

The mask specified in *vec* is not allowed to block SIGKILL or SIGSTOP. The system enforces this restriction silently.

The following is a list of all signals with names in <signal.h> . In this list, those names beginning with an underscore are not defined in POSIX P1003.1 , and are provided as CONVEX extensions.

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction
_SIGTRAP	5*	trace trap
_SIGIOT	6*	IOT instruction

SIGABRT	6*	*abort*(3)
_SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
_SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
_SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe or other socket with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
_SIGURG	16•	urgent condition present on socket
SIGSTOP	17†	stop (cannot be caught, blocked, or ignored)
SIGTSTP	18†	stop signal generated from keyboard
SIGCONT	19•	continue after stop (cannot be blocked)
SIGCHLD	20•	child status has changed
SIGTTIN	21†	background read attempted from control terminal
SIGTTOU	22†	background write attempted to control terminal
_SIGIO	23•	i/o is possible on a descriptor (see *fcntl*(2))
_SIGXCPU	24	cpu time limit exceeded (see *setrlimit*(2))
_SIGXFSZ	25	file size limit exceeded (see *setrlimit*(2))
_SIGVTALRM		26virtual time alarm (see *setitimer*(2))
_SIGPROF	27	profiling timer alarm (see *setitimer*(2))
_SIGWINCH	28•	window size change
_SIGLOST	29*	resource lost (see *lockd*(8C))
SIGUSR1	30	user-defined signal 1
SIGUSR2	31	user-defined signal 2

Once a signal handler is installed, it remains installed until another *sigaction()* call is made, or an *execve*(2) is performed. The default action for a signal may be reinstated by setting *sa_handler* to SIG_DFL; this default is termination except for signals marked with • or †. Signals marked with • are discarded if the action is SIG_DFL; signals marked with † cause the process to stop. If the process is terminated, a "core image" will be made in the current working directory of the receiving process if the signal is one for which an asterisk appears in the above list *and* the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

The effective group ID and the real group ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

Mode of 0666 modified by the file creation mask (see *umask*(2))

File owner ID that is the same as the effective user ID of the receiving process

File group ID that is the same as the file group ID of the current directory

If *sa_handler* is SIG_IGN, the signal is subsequently ignored, and pending instances of the signal are discarded.

Note: the signals SIGKILL and SIGSTOP cannot be ignored.

The *sigaction*(2) call will block if the signal in *sig* is currently being serviced by another thread in the process. The call will be resumed after the thread servicing the signal returns or enables the signal via the *sigsetmask*(2) system call.

After a *fork*(2) or *vfork*(2), the child inherits all signals, the signal mask, and the signal stack, and the restart/interrupt flags.

The *execve*(2) call resets all caught signals to default action and resets all signals to be caught on the user stack. Ignored signals remain ignored, the signal mask remains the same, and the signal stack state is reset.

The *SA_NOCLDSTOP* bit of the *sa_flags* word of the *sigaction* structure controls generation of SIGCHLD in some circumstances. If *sig* is SIGCHLD and SA_NOCLDSTOP is not set in *sa_flags*, a SIGCHLD signal is generated for the calling process whenever any of its child processes stop; if SA_NOCLDSTOP is set, SIGCHLD is not generated.

CONVEX EXTENSIONS

If a caught signal occurs during certain system calls, the call is normally interrupted; it will return –1 and set *errno* to EINTR. The call can be automatically restarted (the default BSD UNIX behavior) by setting the _SA_RESTARTSYS bit in *sa_flags*. The affected system calls are *read*(2) or *write*(2) on a slow device (such as a terminal or pipe or other socket, but not a file) and during a *wait*(2).

Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special stack.

As an extension, if the _SA_ONSTACK bit is set in *sa_flags*, the system will deliver the signal to the process on a *signal stack*, specified with *sigstack*(2).

An alternate mode of signals can be used by setting the _SA_PARALLEL bit in *sa_flags*. When this bit is set, signals that are masked by the delivery of a signal will not be reflected in the masks returned by *sigblock*(2) and *sigsetmask*(2) system calls. It also changes the action taken upon return of signal handler to unblock signals blocked when the handler was invoked rather then resetting the mask to what it was prior to the signal delivery.

The handler routine can be declared:

```
void handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here, *sig* is the signal number into which the hardware faults and traps are mapped as defined below. *code* is a parameter that further defines the type of hardware exception that occurred. *scp* is a pointer to the *sigcontext* structure (defined in <signal.h>) , used to restore the context from before the signal.

The following list defines the mapping of hardware traps to signals and codes. All of these symbols are defined in <signal.h>:

Hardware condition	Signal	Code
Arithmetic traps:		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by zero	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Reserved Operand trap	SIGFPE	FPE_RESOP_TRAP
Segmentation Violations:		
Read access violation	SIGSEGV	SEG_READ_TRAP
Write access violation	SIGSEGV	SEG_WRITE_TRAP
Execute access violation	SIGSEGV	SEG_EXEC_TRAP
Invalid segment	SIGSEGV	SEG_INVSDR_TRAP
Invalid page table page	SIGSEGV	SEG_INVPTP_TRAP
Invalid memory reference	SIGSEGV	SEG_INVDATA_TRAP
I/O access violation	SIGSEGV	SEG_IOACC_TRAP

Ring Violations:		
Inward address reference	SIGBUS	BUS_INWADDR_TRAP
Outward ring call	SIGBUS	BUS_OUTCALL_TRAP
Inward ring return	SIGBUS	BUS_INWRTN_TRAP
Invalid syscall gate	SIGBUS	BUS_INVGATE_TRAP
Invalid return frame length	SIGBUS	BUS_INVFRL_TRAP
Illegal instruction:		
Error exit instruction	SIGILL	ILL_ERRXIT_TRAP
Privileged instruction	SIGILL	ILL_PRIVIN_TRAP
Undefined op code	SIGILL	ILL_UNDFOP_TRAP
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	

RETURN VALUE

A 0 value indicated that the call succeeded. A –1 return value indicates an error occurred and *errno* is set to indicated the reason.

ERRORS

sigaction() will fail and no new signal handler will be installed if one of the following occurs:

[EFAULT] Either *vec* or *ovec* points to memory that is not a valid part of the process address space.

[EINVAL] *sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

[EINTR] The caller is multi-threaded, and while waiting for another thread to complete processing signal *sig* , a distinct, non-blocked signal arrives.

BACKWARD COMPATIBILITY

The *sigaction()* function replaces the *sigvec()* function.

The *struct sigaction* replaces the *struct sigvec.*

Signal handling functions formerly returned type *int*; they now return type *void.* This may be the source of many compile-time warnings.

For non-POSIX compliant calling processes, the _SA_RESTARTSYS bit of *sa_flags* is ignored. Instead, system calls are restarted by default, as they traditionally were under BSD UNIX. In this case, the _SA_INTERRUPT bit may be set to cause system calls *not* to be restarted.

Just as _SA_RESTARTSYS is ignored for non-POSIX callers, the _SA_INTERRUPT bit is ignored for POSIX callers.

In previous releases of the operating system, it was illegal to set the disposition of SIGCONT to SIG_IGN; this is now permitted but has no effect.

SEE ALSO

kill(1), pattach(2), kill(2), sigblock(2), sigsetmask(2), sigpause(2) sigstack(2), setjmp(3), tty(4)

NAME

sigblock, sigunblock – block or unblock signals

SYNOPSIS

```
#include <signal.h>

int sigblock(mask)
int mask;

int sigunblock(mask)
int mask;
```

DESCRIPTION

sigblock causes the signals specified in *mask* to be added to the set of signals currently being blocked from delivery. Signal *i* is blocked if the *i*-th bit in *mask* is a 1.

sigunblock removes the signals specified in *mask* from the set of signals currently being blocked from delivery. If bit *i* of mask is set, then bit *i* of the process's signal mask will be cleared.

It is not possible to block SIGKILL or SIGSTOP; this restriction is silently imposed by the system.

The call will block if any other thread in the process is currently servicing a signal contained in mask. Execution of the call will continue when the thread in the signal handler returns or enables the signal via the *sigsetmask*(2) or *sigunblock*(2) system calls. In this case, the call is interruptible; see **RETURN VALUE**, below.

RETURN VALUE

On success, the previous set of masked signals is returned. The call is always successful for single threaded callers.

For multi-threaded callers, the call may return –1 and *errno* be set to **EINTR**, meaning that the calling thread received a signal which was not blocked while awaiting other threads to finish processing signals in the mask. Note that this return value of –1 is not ambiguous, since the bits representing SIGSTOP and SIGKILL can never be set in the mask returned by a successful call.

SEE ALSO

kill(2), sigvec(2), sigsetmask(2), sigpause(2), sigprocmask(3)

NOTES

The preferred mechanism for signal blocking is *sigprocmask()*.

The *sigunblock()* call is a CONVEX extension to allow reliable processing of parallel signals.

NAME

sigpause – atomically release blocked signals and wait for interrupt

SYNOPSIS

sigpause(sigmask)
int sigmask;

DESCRIPTION

sigpause assigns *sigmask* to the set of masked signals and then waits for a signal to arrive; on return, the set of masked signals is restored for processes that do not use the SV_PARALLEL option of the *sigvec* system call. *sigmask* is usually 0 to indicate that no signals are now to be blocked. *sigpause* always terminates by being interrupted and returns EINTR. The call will block if any other thread in the process is currently running a signal handler for any signal included in *sigmask*. Execution of the call will continue whenever the thread in the signal handler returns or makes a *sigsetmask* call to re-enable the signal.

In normal usage, a signal is blocked using *sigblock*(2). To begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using *sigpause* with the mask returned by *sigblock*.

SEE ALSO

sigblock(2), sigvec(2), sigsetmask(2), kill(2), sigsuspend(3)

NOTES

The preferred mechanism is now *sigsuspend(3)*.

NAME

sigpending – examine pending signals

SYNOPSIS

```
#include <signal.h>

int sigpending(set)
sigset_t *set;
```

DESCRIPTION

sigpending stores the set of signals that are blocked from delivery and pending into the location pointed to by *set*.

RETURN VALUE

Sigpending always succeeds and returns a value of zero.

SEE ALSO

kill(1), sigaction(2), sigprocmask(3)

NAME

sigsetmask – set current signal mask

SYNOPSIS

sigsetmask(mask);
int mask;

DESCRIPTION

Sigsetmask sets the current signal mask (those signals which are blocked from delivery). Signal *i* is blocked if the *i*-th bit in *mask* is a 1.

The system quietly disallows SIGKILL and SIGSTOP to be blocked.

The call will block if any other thread in the process is currently servicing a signal contained in mask. Execution of the call will continue when the thread in the signal handler returns or enables the signal via the *sigsetmask*(2) or *sigunblock*(2) system calls. In this case, the call is interruptible; see **RETURN VALUE**, below.

RETURN VALUE

On success, the previous set of masked signals is returned. The call is always successful for single threaded callers.

For multi-threaded callers, the call may return –1 and *errno* be set to **EINTR**, meaning that the calling thread received a signal which was not blocked while awaiting other threads to finish processing signals in the mask. Note that this return value of –1 is not ambiguous, since the bits representing SIGSTOP and SIGKILL can never be set in the mask returned by a successful call.

SEE ALSO

kill(2), sigvec(2), sigblock(2), sigpause(2)

NAME

sigstack – set and/or get signal stack context

SYNOPSIS

```
#include <signal.h>

struct sigstack {
        caddr_t         ss_sp;
        int     ss_onstack;
};

sigstack(ss, oss);
struct sigstack *ss, *oss;
```

DESCRIPTION

Sigstack allows users to define an alternate stack on which signals are to be processed. If *ss* is non-zero, it specifies a *signal stack* on which to deliver signals and tells the system if the process is currently executing on that stack. When a signal's action indicates its handler should execute on the signal stack (specified with a *sigvec*(2) call), the system checks to see if the process is currently executing on that stack. If the process is not currently executing on the signal stack, the system arranges a switch to the signal stack for the duration of the signal handler's execution. If *oss* is non-zero, the current signal stack state is returned.

NOTES

Signal stacks are not "grown" automatically, as is done for the normal stack. If the stack overflows unpredictable results may occur.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

Sigstack will fail and the signal stack context will remain unchanged if one of the following occurs.

[EFAULT] Either *ss* or *oss* points to memory which is not a valid part of the process address space.

SEE ALSO

sigvec(2), setjmp(3)

NAME

sigvec – software signal facilities

SYNOPSIS

```
#include <signal.h>

struct sigvec {
        int     (*sv_handler)();
        int     sv_mask;
        int     sv_flags;
};

sigvec(sig, vec, ovec)
int sig;
struct sigvec *vec, *ovec;
```

DESCRIPTION

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a *handler* to which a signal is delivered, or specify that a signal is to be *blocked* or *ignored*. A process may also specify that a default action is to be taken by the system when a signal occurs. Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special *signal stack*.

All signals have the same *priority*. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. A global *signal mask* defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It may be changed with a *sigblock*(2) or *sigsetmask*(2) call, or when a signal is delivered to the process.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself.

When a signal is delivered to a process a new signal mask is installed for the duration of the process' signal handler (or until a *sigblock* or *sigsetmask* call is made). This mask is formed by taking the current signal mask, adding the signal to be delivered, and *or*'ing in the signal mask associated with the handler to be invoked.

sigvec assigns a handler for a specific signal. If *vec* is non-zero, it specifies a handler routine and mask to be used when delivering the specified signal. Further, if the SV_ONSTACK bit is set in *sv_flags*, the system will deliver the signal to the process on a *signal stack*, specified with *sigstack*(2). If *ovec* is non-zero, the previous handling information for the signal is returned to the user.

The mask specified in *vec* is not allowed to block SIGKILL, SIGSTOP, or SIGCONT. The system enforces this restriction silently.

The following is a list of all signals with names as in the include file *<signal.h>*:

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction
SIGTRAP	5*	trace trap

SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe or other socket with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGURG	16•	urgent condition present on socket
SIGSTOP	17†	stop (cannot be caught, blocked, or ignored)
SIGTSTP	18†	stop signal generated from keyboard
SIGCONT	19•	continue after stop (cannot be blocked)
SIGCHLD	20•	child status has changed
SIGTTIN	21†	background read attempted from control terminal
SIGTTOU	22†	background write attempted to control terminal
SIGIO	23•	i/o is possible on a descriptor (see *fcntl*(2))
SIGXCPU	24	cpu time limit exceeded (see *setrlimit*(2))
SIGXFSZ	25	file size limit exceeded (see *setrlimit*(2))
SIGVTALRM	26	virtual time alarm (see *setitimer*(2))
SIGPROF	27	profiling timer alarm (see *setitimer*(2))
SIGWINCH	28•	window size change
SIGLOST	29*	resource lost (see *lockd*(8C))
SIGUSR1	30	user-defined signal 1
SIGUSR2	31	user-defined signal 2

The starred signals in the list above cause a core image if not caught or ignored.

Once a signal handler is installed, it remains installed until another *sigvec* call is made, or an *execve*(2) is performed. The default action for a signal may be reinstated by setting *sv_handler* to SIG_DFL; this default is termination except for signals marked with • or †. Signals marked with • are discarded if the action is SIG_DFL; signals marked with † cause the process to stop. If the process is terminated, a "core image" will be made in the current working directory of the receiving process if the signal is one for which an asterisk appears in the above list *and* the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

The effective group ID and the real group ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask (see *umask*(2))

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the file group ID of the current directory

If *sv_handler* is SIG_IGN the signal is subsequently ignored and pending instances of the signal are discarded.

Note: the signals SIGKILL, SIGSTOP, and SIGCONT cannot be ignored.

If a caught signal occurs during certain system calls, the call is normally restarted. The call can be forced to terminate prematurely with an EINTR error return by setting the SV_INTERRUPT bit in *sv_flags*. The affected system calls are *read*(2) or *write*(2) on a slow device (such as a terminal or pipe or other socket, but not a file) and during a *wait*(2).

An alternate mode of signals can be used by setting the SV_PARALLEL bit in sv_flags. When this bit is set, signals that are masked by the delivery of a signal will not be reflected in the masks returned by *sigblock*(2) and *sigsetmask*(2) system calls. It also changes the action taken upon return of signal handler to unblock signals blocked when the handler was invoked rather then resetting the mask to what it was prior to the signal delivery.

The *sigvec*(2) call will block if the signal in *sig* is currently being serviced by another thread in the process. The call will be resumed after the thread servicing the signal returns or enables the signal via the *sigsetmask*(2) system call.

After a *fork*(2) or *vfork*(2) the child inherits all signals, the signal mask, and the signal stack, and the restart/interrupt flags.

The *execve*(2) call resets all caught signals to default action and resets all signals to be caught on the user stack. Ignored signals remain ignored; the signal mask remains the same; the signal stack state is reset.

NOTES

The handler routine can be declared:

```
handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here *sig* is the signal number, into which the hardware faults and traps are mapped as defined below. *code* is a parameter which further defines the type of hardware exception which occurred. *scp* is a pointer to the *sigcontext* structure (defined in <*signal.h*>), used to restore the context from before the signal.

The following defines the mapping of hardware traps to signals and codes. All of these symbols are defined in <*signal.h*>:

Hardware condition	Signal	Code
Arithmetic traps:		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by zero	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Reserved Operand trap	SIGFPE	FPE_RESOP_TRAP
Segmentation Violations:		
Read access violation	SIGSEGV	SEG_READ_TRAP
Write access violation	SIGSEGV	SEG_WRITE_TRAP
Execute access violation	SIGSEGV	SEG_EXEC_TRAP
Invalid segment	SIGSEGV	SEG_INVSDR_TRAP
Invalid page table page	SIGSEGV	SEG_INVPTP_TRAP
Invalid memory reference	SIGSEGV	SEG_INVDATA_TRAP
I/O access violation	SIGSEGV	SEG_IOACC_TRAP
Ring Violations:		
Inward address reference	SIGBUS	BUS_INWADDR_TRAP
Outward ring call	SIGBUS	BUS_OUTCALL_TRAP
Inward ring return	SIGBUS	BUS_INWRTN_TRAP
Invalid syscall gate	SIGBUS	BUS_INVGATE_TRAP
Invalid return frame length	SIGBUS	BUS_INVFRL_TRAP
Illegal instruction:		
Error exit instruction	SIGILL	ILL_ERRXIT_TRAP
Privileged instruction	SIGILL	ILL_PRIVIN_TRAP

Undefined op code	SIGILL	ILL_UNDFOP_TRAP
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	

RETURN VALUE

A 0 value indicated that the call succeeded. A –1 return value indicates an error occurred and *errno* is set to indicated the reason.

ERRORS

sigvec will fail and no new signal handler will be installed if one of the following occurs:

[EFAULT] Either *vec* or *ovec* points to memory which is not a valid part of the process address space.

[EINVAL] *Sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

[EINVAL] An attempt is made to ignore SIGCONT (by default SIGCONT is ignored).

SEE ALSO

kill(1), pattach(2), kill(2), sigblock(2), sigsetmask(2), sigpause(2) sigstack(2), setjmp(3), tty(4)

NAME

socket – create an endpoint for communication

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>

s = socket(domain, type, protocol)
int s, domain, type, protocol;

DESCRIPTION

Socket creates an endpoint for communication and returns a descriptor.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file *<sys/socket.h>*. The currently understood formats are

PF_UNIX or AF_UNIX (UNIX internal protocols),
PF_INET or AF_INET (ARPA Internet protocols),
PF_NS or AF_NS (Xerox Network Systems protocols), and
PF_IMPLINK or AF_IMPLINK (IMP "host at IMP" link layer).

The socket has the indicated *type,* which specifies the semantics of communication. Currently defined types are:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). A SOCK_SEQPACKET socket may provide a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer may be required to read an entire packet with each read system call. This facility is protocol specific, and presently implemented only for PF_NS. SOCK_RAW sockets provide access to internal network protocols and interfaces. The types SOCK_RAW, which is available only to the super-user, and SOCK_RDM, which is planned, but not yet implemented, are not described here.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the "communication domain" in which communication is to take place; see *protocols*(3N).

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a *connect*(2) call. Once connected, data may be transferred using *read*(2) and *write*(2) calls or some variant of the *send*(2) and *recv*(2) calls. When a session has been completed a *close*(2) may be performed. Out-of-band data may also be transmitted as described in *send*(2) and received as described in *recv*(2).

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with –1 returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets "warm" by forcing transmissions

roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that *read*(2) calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in *send*(2) calls. Datagrams are generally received with *recvfrom*(2), which returns the next datagram with its return address.

An *fcntl*(2) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives. It may also enable non-blocking I/O and asynchronous notification of I/O events via SIGIO.

The operation of sockets is controlled by socket level *options*. These options are defined in the file *<sys/socket.h>*. *Setsockopt*(2) and *getsockopt*(2) are used to set and get options, respectively.

RETURN VALUE

A –1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

ERRORS

The *socket* call fails if:

[EPROTONOSUPPORT] The protocol type or the specified protocol is not supported within this domain.

[EAFNOSUPPORT] The address family is not supported within this domain. Ordinarily equivalent to EPROTONOSUPPORT.

[EMFILE] The per-process descriptor table is full.

[ENFILE] The system file table is full.

[EACCESS] Permission to create a socket of the specified type and/or protocol is denied.

[ENOBUFS] Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

SEE ALSO

accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), read(2), recv(2), select(2), send(2), shutdown(2), socketpair(2), write(2)

NOTES

The PF_NS and PF_IMPLINK protocol families are currently unsupported.

NAME

socketpair – create a pair of connected sockets

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

socketpair(d, type, protocol, sv)
int d, type, protocol;
int sv[2];
```

DESCRIPTION

The *socketpair* call creates an unnamed pair of connected sockets in the specified domain *d*, of the specified *type*, and using the optionally specified *protocol*. The descriptors used in referencing the new sockets are returned in *sv*[0] and *sv*[1]. The two sockets are indistinguishable.

DIAGNOSTICS

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

The call succeeds unless:

[EMFILE]	Too many descriptors are in use by this process.
[EAFNOSUPPORT]	The specified address family is not supported on this machine.
[EPROTONOSUPPORT]	The specified protocol is not supported on this machine.
[EFAULT]	The address *sv* does not specify a valid part of the process address space.

SEE ALSO

read(2), write(2), pipe(2)

BUGS

This call is currently implemented only for the UNIX domain.

NAME

stat, fstat – get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

stat(path, buf)
char *path;
struct stat *buf;

fstat(fd, buf)
int fd;
struct stat *buf;
```

DESCRIPTION

stat obtains information about the file named by *path*. Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

fstat obtains the same information about an open file referenced by the argument descriptor, such as would be obtained by an *open* call.

buf is a pointer to a *stat* structure into which information is placed concerning the file. The contents of the structure pointed to by *buf* include the following members:

```
struct stat {
        dev_t     st_dev;     /* device inode resides on */
        ino_t     st_ino;     /* this inode's number */
        mode_t    st_mode;    /* protection */
        nlink_t   st_nlink;   /* number of hard links to the file */
        uid_t     st_uid;     /* user-id of owner */
        gid_t     st_gid;     /* group-id of owner */
        off_t     st_size;    /* total size of file */
        time_t    st_atime;   /* file last access time */
        time_t    st_mtime;   /* file last modify time */
        time_t    st_ctime;   /* file last status change time */
        /* The following are CONVEX extensions */
        dev_t     st_rdev;    /* the device type, for inode that is device */
        long      st_blksize; /* optimal blocksize for file system i/o ops */
        long      st_blocks;  /* actual number of blocks allocated */
};
```

st_atime Time when file data was last read or modified. Changed by the following system calls: *mknod*(2), *utimes*(2), *read*(2), *write*(2), and *truncate*(2). For reasons of efficiency, st_atime is not set when a directory is searched though this would be more logical.

st_mtime Time when data was last modified. It is not set by changes of owner, group, link count, or mode. Changed by the following system calls: *mknod*(2), *truncate*(2), *utimes*(2), *write*(2).

st_ctime Time when file status was last changed. It is set both both by writing and changing the inode. Changed by the following system calls: *chmod*(2) *chown*(2), *link*(2), *mknod*(2), *rename*(2), *unlink*(2), *utimes*(2), *write*(2), *truncate*(2).

st_mode The attributes of st_mode may be queried with the following macros:

S_ISDIR(st.st_mode) True if a directory

S_ISCHR(st.st_mode) True if a character special device

S_ISBLK(st.st_mode) True if a block special device

S_ISREG(st.st_mode) True if a regular file

S_ISFIFO(st.st_mode) True if a pipe or fifo special file

The permissions associated with st_mode may be extracted with the S_IRWXU, S_IRWXG, and S_IRWXO masks. The set-user-ID bit is S_ISUID and the set-group-ID bit is S_ISGID. Refer to *chmod*(2) for more details.

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

stat will fail if one or more of the following are true:

[ENOENT] *path* points to an empty string.

[ENOTDIR] A component of the path prefix of *path* is not a directory.

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG]
The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters. The file referred to by *path* does not exist.

[EACCES] Search permission is denied for a component of the path prefix of *path*.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[EFAULT] *buf* or *path* points to an invalid address.

[EIO] An I/O error occurred while reading from or writing to the file system.

fstat will fail if one or both of the following are true:

[EBADF] *fd* is not a valid open file descriptor.

[EFAULT] *buf* points to an invalid address.

[EIO] An I/O error occurred while reading from or writing to the file system.

BACKWARD COMPATIBLITY

In previous versions of the operating system, the empty pathname string was a synonym for the current directory.

The file status information was previously expressed as follows:

```
#define  S_IFMT     0170000   /* type of file */
#define  S_IFIFO    0010000   /* fifo special */
#define  S_IFCHR    0020000   /* character special */
#define  S_IFDIR    0040000   /* directory */
#define  S_IFBLK    0060000   /* block special */
#define  S_IFREG    0100000   /* regular */
#define  S_IFLNK    0120000   /* symbolic link */
#define  S_IFSOCK   0140000   /* socket */
#define  S_ISUID    0004000   /* set user id on execution */
#define  S_ISGID    0002000   /* set group id on execution */
#define  S_ISVTX    0001000   /* see sticky(8) */
#define  S_IREAD    0000400   /* read permission, owner */
#define  S_IWRITE   0000200   /* write permission, owner */
```

```
#define S_IEXEC 0000100 /* execute/search permission, owner */
```

The mode bits 0000070 and 0000007 encode group and others permissions (see *chmod*(2)).

SEE ALSO

chmod(2), chown(2), lstat(2), readlink(2), utimes(2)

NOTES

Applying *fstat* to a socket (and thus to a pipe) returns a zero'd buffer, except for the blocksize field, and a unique device and inode number.

NAME

statfs, fstatfs – get file system statistics

SYNOPSIS

```
#include <sys/types.h>
#include <sys/vfs.h>
#include <sys/mount.h>

statfs(path, buf)
char *path;
struct statfs *buf;

fstatfs(fd, buf)
int fd;
struct statfs *buf;
```

DESCRIPTION

statfs returns information about a mounted file system. *path* is the path name of any file within the mounted filesystem. *buf* is a pointer to a *statfs* structure defined as follows:

```
typedef struct {
        long    val[2];
} fsid_t;

struct statfs {
        long    f_type;       /* filesystem type, MOUNT_UFS or MOUNT_NFS */
        long    f_bsize;      /* fundamental file system block size */
        long    f_blocks;     /* total blocks in file system */
        long    f_bfree;      /* free blocks */
        long    f_bavail;     /* free blocks available to non-superuser */
        long    f_files;      /* total file nodes in file system */
        long    f_ffree;      /* free file nodes in fs */
        fsid_t  f_fsid;       /* file system ID */
        long    f_spare[7];   /* spare for later */
};
```

Fields that are undefined for a particular file system are set to –1. (Note that *f_bsize* is the frag size for 4.2 type file systems. The block size can be obtained from *fstat*.) *fstatfs* returns the same information about an open file referenced by descriptor *fd*.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, –1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

statfs fails if one or more of the following are true:

[ENOTDIR] A component of the path prefix of *path* is not a directory.

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG] The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.

[ENOENT] The file referred to by *path* does not exist.

[EACCES] Search permission is denied for a component of the path prefix of *path*.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[EFAULT] *buf* or *path* points to an invalid address.

[EIO] An I/O error occurred while reading from or writing to the file system.

fstatfs fails if one or both of the following are true:

[EBADF]	*fd* is not a valid open file descriptor.
[EFAULT]	*buf* points to an invalid address.
[EIO]	An I/O error occurred while reading from or writing to the file system.

NAME

swapon – add a swap device for interleaved paging/swapping

SYNOPSIS

swapon(special)
char *special;

DESCRIPTION

Swapon makes the block device *special* available to the system for allocation for paging and swapping. The names of potentially available devices are known to the system and defined at system configuration time, however, these defaults may be modified at boot time. The size of the swap area on *special* is calculated at the time the device is first made available for swapping.

RETURN VALUE

If successful, swapon returns 0. If unsuccessful, a –1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

swapon will fail if one or more of the following is true:

[EPERM] The *swapon* system call is only valid when issued by a superuser. Since the list of valid swap devices often includes partitions which are being used for normal file systems, and swapping on such a file system would destroy the data on that file system, ordinary users are not permitted to use *swapon.*

[ENOTBLK] The specified swap device is not a block device.

[EBUSY] The specified swap device is all ready in use.

[EINVAL] The specified swap device is not included in the list of valid swap devices. The default list is set by the *sysgen* utility and includes most "b" partitions but can be modified at boot time.

SEE ALSO

swapon(8) sysgen(8)
"Boot-Time Parameters" chapter of the CONVEX System Manager's Guide.

CAVEATS

Superusers should be careful not to swap on a partition unless they are willing to lose all existing data on that partition, and should *never* swap on a mounted file system.

BUGS

There is no way to stop swapping on a disk so that the pack may be dismounted.

NAME

symlink – make symbolic link to a file

SYNOPSIS

symlink(name1, name2)
char *name1, *name2;

DESCRIPTION

A symbolic link *name2* is created to *name1* (*name2* is the name of the file created, *name1* is the string used in creating the symbolic link). Either name may be an arbitrary path name; the files need not be on the same file system.

RETURN VALUE

Upon successful completion, a zero value is returned. If an error occurs, the error code is stored in *errno* and a –1 value is returned.

ERRORS

The symbolic link is made unless on or more of the following are true:

[EINVAL]	Either *name1* or *name2* contains a character with the high-order bit set.
[ENOENT]	One of the pathnames specified was too long.
[ENOTDIR]	A component of the *name2* prefix is not a directory.
[EEXIST]	*Name2* already exists.
[EACCES]	A component of the *name2* path prefix denies search permission.
[EROFS]	The file *name2* would reside on a read-only file system.
[EFAULT]	*Name1* or *name2* points outside the process's allocated address space.
[ELOOP]	Too may symbolic links were encountered in translating the pathname.

SEE ALSO

link(2), ln(1), unlink(2)

NAME

sync – update super-block

SYNOPSIS

sync()

DESCRIPTION

Sync causes all information in core memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

Sync should be used by programs which examine a file system, for example *fsck, df,* etc. *Sync* is mandatory before a boot.

SEE ALSO

fsync(2), sync(8), update(8)

BUGS

The writing, although scheduled, is not necessarily complete upon return from *sync*.

NAME

syscall – indirect system call

SYNOPSIS

syscall(number, arg, ...)

DESCRIPTION

Syscall performs the system call whose assembly language interface has the specified *number* and arguments *arg*.

The s0 return value of the system call is returned. See /usr/include.

DIAGNOSTICS

Syscall returns -1 and sets the external variable *errno* (see intro(2)) when the call returns unsuccessfully.

NAME

truncate, ftruncate – truncate a file to a specified length

SYNOPSIS

truncate(path, length)
char *path;
int length;

ftruncate(fd, length)
int fd, length;

DESCRIPTION

truncate causes the file named by *path* or referenced by *fd* to be truncated to at most *length* bytes in size. If the file previously was larger than this size, the extra data is lost. With *ftruncate*, the file must be open for writing.

RETURN VALUES

A value of 0 is returned if the call succeeds. If the call fails a –1 is returned, and the global variable *errno* specifies the error.

ERRORS

truncate succeeds unless:

[EINVAL]	The pathname contains a character with the high-order bit set.
[ENOENT]	The pathname was too long.
[ENOTDIR]	A component of the path prefix of *path* is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	A component of the *path* prefix denies search permission.
[EISDIR]	The named file is a directory.
[EROFS]	The named file resides on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EFAULT]	*name* points outside the process's allocated address space.

ftruncate succeeds unless:

[EBADF]	The *fd* is not a valid descriptor.
[EINVAL]	The *fd* references a socket, not a file.

SEE ALSO

open(2)

BUGS

Partial blocks discarded as the result of truncation are not zero filled; this can result in holes in files which do not read as zero.

These calls should be generalized to allow ranges of bytes in a file to be discarded.

NAME

umask – set file creation mode mask

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

mode_t umask(numask)
mode_t numask;
```

DESCRIPTION

umask sets the process's file mode creation mask to *numask* and returns the previous value of the mask. Only the file permission bits of cmask are used, as defined by the inclusive OR of S_IRWXU, S_IRWXG, and S_IRWXO (see *stat*(2)). The permission bits of *numask* are used whenever a file is created, clearing corresponding bits in the file mode (see *chmod*(2)). This clearing allows each user to restrict the default access to his files.

The value is initially (S_IWGRP|S_IWOTH), allowing write access for the file's owner only. The mask is inherited by child processes.

RETURN VALUE

The previous value of the file mode mask is returned by the call.

SEE ALSO

chmod(2), mknod(2), open(2), stat(2)

NAME

uname – get system information

SYNOPSIS

#include <sys/utsname.h>

uname(struct utsname *namep)

DESCRIPTION

Uname stores information identifying the operating system and execution environment in the structure pointed to by *namep*. It includes the functionality of the *getsysinfo(2)* and *gethostname(2)* system calls of ConvexOS, although the old calls remain for backward compatibility.

The *utsname* structure will be filled in with the following information:

```
#define _UTS_NMLN    64

        struct  utsname {
                struct  {
                        unsigned  short     system_sn;
                        unsigned  char      cpu_type;
                        unsigned  char      cpu_count;
                        unsigned  long      flags;
                        unsigned  long      flags2;
                }       sysinfo;
                char    sysname[_UTS_NMLN];
                char    nodename[_UTSNMLN];
                char    release[_UTSNMLN];
                char    version[UTS_NMLN];
                char    machine[_UTSNMLN];
        };
```

The fields of *sysinfo* are interpreted as follows:

system_sn — Serial number of the system on which the program is running.

cpu_type — Indicates the CPU type of the CPU(s) in the complex.

cpu_count — Indicates the number of CPU(s) in the complex. This duplicates the information in third ASCII digit of the *release* field described below.

flags — Various flags indicating hardware and software capabilities. See the header file sys/utsname.h for detailed descriptions of the flags.

flags2 — Currently unused.

All remaining fields are null terminated character arrays.

sysname — Name of this implementation of the operating system, e.g. "vmunix".

nodename — Last host name configured with *sethostname(2)*; this is the same information as returned by *gethostname(2)*.

release — The configuration of the machine, e.g. "C210". Specifically, the first character is 'C', the second is '1' or '2'. The third is the number of processors, and the fourth is always '0'.

version — The version of ConvexOS, e.g. "8.0".

machine — Always returns "convex".

RETURN VALUE

If the call succeeds, a value of 0 is returned. If the call fails, then a value of –1 is returned and an error code is placed in the global location *errno*.

ERRORS

[EFAULT] The pointer *namep* is invalid.

SEE ALSO

getsysinfo(2), gethostname(2), sethostname(2)

NAME

remove, unlink – remove a file or directory entry

SYNOPSIS

int unlink(char *path);

#include <stdio.h>
int remove(const char *path);

DESCRIPTION

unlink removes the entry for the file *path* from its directory. If this entry was the last link to the file and no process has the file open, then all resources associated with the file are reclaimed. However, if the file was open in any process, the actual resource reclamation is delayed until it is closed even though the directory entry has disappeared.

remove performs the same function as *unlink*.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a nonzero value is returned and *errno* is set to indicate the error.

ERRORS

The *unlink* succeeds unless:

[EINVAL]	The path contains a character with the high-order bit set.
[ENAMETOOLONG]	The pathname is too long.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[ENOENT]	The *path* argument points to the empty string.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EPERM]	The named file is a directory and the effective user ID of the process is not the super-user.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

BACKWARD COMPATIBILITY

Previous versions of the operating system allowed the empty path name as a synonym for the current directory.

SEE ALSO

close(2), link(2), rmdir(2)

NAME

unmount – remove a file system

SYNOPSIS

unmount(name)
char *name;

DESCRIPTION

unmount announces to the system that the directory *name* is no longer to refer to the root of a mounted file system. The directory *name* reverts to its ordinary interpretation.

RETURN VALUE

unmount returns 0 if the action occurred; –1 if if the directory is inaccessible or does not have a mounted file system, or if there are active files in the mounted file system.

ERRORS

unmount may fail with one of the following errors:

[EPERM]	The caller is not the superuser.
[EINVAL]	*name* is not the root of a mounted file system.
[EBUSY]	A process is holding a reference to a file located on the file system.
[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	The pathname was too long.
[ENOENT]	*name* does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*name* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EIO]	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

mount(2), mount(8), umount(8)

BUGS

The error codes are in a state of disarray; too many errors appear to the caller as one value.

NAME

utimes – set file times

SYNOPSIS

```
#include <sys/time.h>

utimes(file, tvp)
char *file;
struct timeval *tvp[2];
```

DESCRIPTION

The *utimes* call uses the "accessed" and "updated" times in that order from the *tvp* vector to set the corresponding recorded times for *file*. If *tvp* is null, the current time is used.

The caller must be the owner of the file or the super-user. The "inode-changed" time of the file is set to the current time.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

utimes will fail if one or more of the following are true:

[EINVAL]	The pathname contained a character with the high-order bit set.
[ENOENT]	The pathname was too long.
[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EACCES]	A component of the path prefix denies search permission.
[EPERM]	The process is not super-user and not the owner of the file.
[EACCES]	The effective user ID is not super-user and not the owner of the file, *times* is NULL, and write access is denied.
[EROFS]	The file resides in a file system which is mounted read only.
[EFAULT]	*tvp* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

SEE ALSO

stat(2)

NAME

vadvise – give advice to paging system

SYNOPSIS

#include <sys/vadvise.h>

vadvise(param)
int param;

DESCRIPTION

vadvise is used to inform the system that process paging behavior merits special consideration. Parameters to *vadvise* are defined in the file *<sys/vadvise.h>*:

VA_ANOM advises that the paging behavior is not likely to be well handled by the system's default algorithm, since reference information is collected over macroscopic intervals (e.g. 10-20 seconds) will not serve to indicate future page references. The system in this case will choose to replace pages with little emphasis placed on recent usage. It is essential that processes which have very random paging behavior (such as LISP during garbage collection of very large address spaces) call *vadvise,* as otherwise the system has great difficulty dealing with their page-consumptive demands.

VA_SEQL advises that page access is sequential. This is useful, for example, when iterating over the elements of a large array.

VA_NORM restores default paging behavior.

VA_FLUSH invalidates all the process' pages. vadvise(VA_FLUSH) should be called when the set of pages the process is referencing is going to change greatly.

BUGS

Will go away soon, being replaced by a per-page *madvise* facility.

NAME

vfork – spawn new process in a virtual memory efficient way

SYNOPSIS

pid = vfork()
int pid;

DESCRIPTION

Vfork can be used to create new processes without fully copying the address space of the old process, which is horrendously inefficient in a paged environment. It is useful when the purpose of *fork*(2) would have been to create a new system context for an *execve*. *Vfork* differs from *fork* in that the child borrows the parent's memory and thread of control until a call to *execve*(2) or an exit (either by a call to *exit*(2) or abnormally.) The parent process is suspended while the child is using its resources.

Vfork returns 0 in the child's context and (later) the pid of the child in the parent's context.

Vfork can normally be used just like *fork*. It does not work, however, to return while running in the childs context from the procedure which called *vfork* since the eventual return from *vfork* would then return to a no longer existent stack frame. Be careful, also, to call *_exit* rather than *exit* if you can't *execve*, since *exit* will flush and close standard I/O channels, and thereby mess up the parent processes standard I/O data structures. (Even with *fork* it is wrong to call *exit* since buffered data would then be flushed twice.)

SEE ALSO

fork(2), execve(2), sigvec(2), wait(2),

DIAGNOSTICS

Same as for *fork*.

BUGS

This system call will be eliminated when proper system sharing mechanisms are implemented. Users should not depend on the memory sharing semantics of *vfork* as it will, in that case, be made synonymous to *fork*.

To avoid a possible deadlock situation, processes which are children in the middle of a *vfork* are never sent SIGTTOU or SIGTTIN signals; rather, output or *ioctl*s are allowed and input attempts result in an end-of-file indication.

NAME

vhangup – virtually "hangup" the current control terminal

SYNOPSIS

vhangup()

DESCRIPTION

Vhangup is used by the initialization process *init*(8) (among others) to arrange that users are given "clean'" terminals at login, by revoking access of the previous users' processes to the terminal. To effect this, *vhangup* searches the system tables for references to the control terminal of the invoking process, revoking access permissions on each instance of the terminal which it finds. Further attempts to read from the terminal will yield a return value of 0, as if end-of-file had been detected. Further attempts to write or otherwise access the terminal by the affected processes will yield i/o errors (EIO). Finally, a hangup signal (SIGHUP) is sent to the process group of the control terminal.

SEE ALSO

init (8)

BACKWARD COMPATIBILITY

In previous releases of the operating system, attempts to access a 'vhungup' terminal resulted in an *errno* of EBADF.

In previous releases of the operating system, access to the control terminal via **/dev/tty** was still possible; this is no longer the case.

NOTES

This call has been obsoleted by an automatic mechanism which takes place on process exit.

NAME

wait, waitpid – wait for process to terminate

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(status)
int *status;

pid_t waitpid(pid, status, options)
pid_t pid;
int *status;
int  options;
```

DESCRIPTION

wait causes its caller to delay until a signal is received or one of its child processes terminates. If any child has died since the last call to *wait*, return is immediate, returning the process ID and exit status of one of the terminated children. If there are no children, return is immediate with the value –1 returned.

The *waitpid()* function behaves identically to *wait()* if the *pid* argument is –1 and the *options* argument is zero. Otherwise, *pid* specifies a set of child processes for which status is requested. If *pid* is –1, status is desired for any child. If *pid* is greater than zero, it specifies a single child for which status is desired. If *pid* is zero, status is requested for any process whose process group ID is equal to that of the caller. If *pid* is less than –1, status is desired for any child whose process group ID is equal to the absolute value of *pid*.

On return from a successful *wait()* or *waitpid()* call, if *status* is not NULL, it is filled in with information revealing the terminated child's cause for exit. Macros are provided in **<sys/wait.h>** to test the exit status.

WIFEXITED(status)
: Evaluates non-zero if status was returned for a child that terminated normally (by calling _exit(2)).

WEXITSTATUS(status)
: If WIFEXITED(status) is non-zero, this macro provides the low-order 8 bits of the argument that the child passed to _exit().

WIFSIGNALED(status)
: Evaluates non-zero if the child terminated due to receipt of a signal that was not caught.

WTERMSIG(status)
: If WIFSIGNALED(status) is non-zero, this macro provides the number of the signal that caused the termination of the child process.

WIFSTOPPED(status)
: Evaluates non-zero if the child is currently stopped.

WSTOPSIG(status)
: If WIFSTOPPED (status) is non-zero, this macro yields the number of the signal that caused the child to stop.

The *options* argument to *waitpid* supports the following values:

WNOHANG
: The caller will not be suspended if no child status is immediately available; in such a case, the return value from *waitpid* will be zero.

WUNTRACED Also reports child processes that are stopped and whose status has not been reported since they stopped.

RETURN VALUE

If *wait* or *waitpid* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. If *waitpid* was called with WUNTRACED set in *options* and it has at least one child specified by *pid*, but status is available for no child specified *pid*, zero will be returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

wait3 and *cvxwait* return –1 if there are no children not previously waited for; 0 is returned if WNOHANG is specified and there are no stopped or exited children.

ERRORS

wait will fail and return immediately if one or more of the following is true:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EFAULT] The *status* or *rusage* arguments point to an illegal address.

Waitpid will fail and return immediately if one or more of the following is true:

[ECHILD] The process or process group specified by *pid* does not exist or is not a child of the calling process.

[EINTR] The function was interrupted by a signal.

[EINVAL] The *options* argument contains bits other than WNOHANG or WUNTRACED.

SEE ALSO

exit(2)

CONVEX EXTENSIONS

CONVEX supplies the following extensions:

```
#include <sys/time.h>
#include <sys/resource.h>

pid = wait3(status, options, rusage)
int pid;
int *status;
int options;
struct rusage *rusage;

pid = cvxwait(status, options, rusage)
int pid;
int *status;
int options;
struct cvxrusage *rusage;
```

wait3 provides an alternate interface for programs that must not block when collecting the status of child processes. The *status* and *options* parameters are defined as above. If *rusage* is non-zero, a summary of the resources used by the terminated process and all its children is returned (this information is currently not available for stopped processes).

The *cvxwait* system call is identical to *wait3* but returns a struct cvxrusage instead of a struct rusage. The primary difference between these is that struct cvxrusage contains information about the level of parallelism of the terminated process.

BACKWARD COMPATIBILITY

Previous releases of the operating system documented use of a pointer to *union wait*, where the current release advocates use of a pointer to integer. To convert existing code, simply change calls of the following type: **union wait wait_union; wait(&wait_union)** to this new idiom: **wait(&wait_union.w_status)** or this one: **wait((int *)&wait_union).** The existing code will

then continue to work as before.

Previous versions of the operating system by default restarted the *wait* family functions when a signal was received while awaiting termination of a child. See *sigaction(2)* for details of controlling this behavior.

NOTES

See *sigaction*(2) for a list of termination statuses (signals); 0 status indicates normal termination. A special status (0177) is returned for a stopped process that has not terminated and can be restarted. See *pattach*(2). If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

If the parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

NAME

write – write output on a file

SYNOPSIS

```
#include <unistd.h>

write(d, buf, nbytes)
int d;
char *buf;
unsigned nbytes;
```

DESCRIPTION

write attempts to write *nbytes* of data to the object referenced by the descriptor *d* from the buffer pointed to by *buf*.

On objects capable of seeking, the *write* starts at a position given by the pointer associated with *d*. See *lseek*(2). Objects that are not capable of seeking always write from the current position. The value of the pointer associated with such an object is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

If the real user is not the super-user, then *write* clears the set-user-ID bit on a file. This prevents penetration of system security by a user who "captures" a writable set-user-ID file owned by the super-user.

When using non-blocking I/O on objects that are subject to flow control, such as sockets, pipes (or FIFOs), or terminals, *write* may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible. If such an object's buffers are full, so that it cannot accept any data, then *write* will return –1 and set *errno* to EAGAIN. Otherwise, *write* will block until space becomes available.

CONVEX EXTENSIONS

write may operate synchronously (default) or asynchronously, depending on the FASIO flag set with the *fcntl* system call. Synchronous writes suspend the caller until the system has processed the write request and return the number of bytes actually written. The file position pointer is incremented by the number of bytes actually written. Asynchronous writes return before the request has been fully processed, and unless there are errors in the arguments passed in, asynchronous writes always return the number of bytes requested and increment the file pointer by the number of bytes requested. Use *asiostat*(2). to determine the completion status of an asynchronous transfer.

RETURN VALUE

Upon successful completion, the number of bytes actually written is returned unless the write was asynchronous, in which case the number of bytes requested to be transferred is returned. If unsuccessful, a –1 is returned and *errno* is set to indicate the error.

ERRORS

write will fail and the file pointer will remain unchanged if one or more of the following are true:

[EAGAIN] The file was marked for non-blocking I/O, and no data could be written immediately.

[EBADF] *d* is not a valid descriptor open for writing.

[EDQUOT] The user's quota of disk blocks on the file system containing the file has been exhausted.

[EFAULT] Part of the data to be written to the file points outside the process's allocated address space.

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or

the maximum file size.

[EINTR] The call is forced to terminate prematurely due to the arrival of a signal whose _SA_INTERRUPT bit in **sa_flags** is set (see *sigaction*(2))

[EIO] The process is in a background process group and attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU signals, and the process group of the process is orphaned. (The most likely scenario is that the user logged off with a job running in the background that attempts to write to the terminal.)

[EINVAL] The requested transfer size is too big for a single request. In order to prevent possible memory deadlocks, the kernel will refuse to do transfers larger than approximately half the size of physical memory.

[EINVAL] The pointer associated with *d* was negative.

[ENOSPC] There is no free space remaining on the file system containing the file.

[EIO] An I/O error occurred while reading from or writing to the file system.

[EPIPE] An attempt is made to write to a pipe that is not open for reading by any process (or to a socket of type SOCK_STREAM that is connected to a peer socket). Note: an attempted write of this kind will also cause you to receive a SIGPIPE signal from the kernel. If you've not made a special provision to catch or ignore this signal, your process will die.

[ENODMON] *Migration only:* The file referenced by *d* is under daemon control, but no daemon is present.

EISMIGRATED *Migration only:* A block in the file reference by *d* is migrated, and the migration daemon was unable to stage the file in from secondary storage.

BACKWARD COMPATIBILITY

Where previous versions of the operating system returned EWOULDBLOCK for *errno*, EAGAIN is now returned.

The default signal behavior is now to interrupt and not restart a *write* operation.

SEE ALSO

asiostat(2), fcntl(2), lseek(2), open(2), pipe(2), select(2)

NAME

writev – gather output to file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>

int writev(d, iov, ioveclen)
int d;
struct iovec *iov;
int ioveclen;
```

DESCRIPTION

writev gathers output data from the *ioveclen* buffers specified by the members of the *iov* array: iov[0], iov[1], ..., iov[iovcnt–1] and writes the data to the file indicated by descriptor *d*.

The *iovec* structure is defined as

```
struct iovec {
        caddr_t iov_base;
        int     iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory from which data should be written. *writev* will always write a complete area before proceeding to the next.

On objects capable of seeking, the *writev* starts at a position given by the pointer associated with *d*. See *lseek*(2). Objects that are not capable of seeking always write from the current position. The value of the pointer associated with such an object is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

writev may operate synchronously (default) or asynchronously, depending on the FASIO flag set with the *fcntl* system call. Synchronous writes suspend the caller until the system has processed the write request and return the number of bytes actually written. The file position pointer is incremented by the number of bytes actually written. Asynchronous writes return before the request has been fully processed, and unless there are errors in the arguments passed in, asynchronous writes always return the number of bytes requested and increment the file pointer by the number of bytes requested. Use *asiostat*(2). to determine the completion status of an asynchronous transfer.

If the real user is not the super-user, then *writev* clears the set-user-ID bit on a file. This prevents penetration of system security by a user who "captures" a writable set-user-ID file owned by the super-user.

When using non-blocking I/O on objects that are subject to flow control, such as sockets, pipes (or FIFOs), or terminals, *writev* may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible. If such an object's buffers are full so that it cannot accept any data, then *writev* will return –1 and set *errno* to EWOULDBLOCK. Otherwise, they will block until space becomes available.

RETURN VALUE

Upon successful completion, the number of bytes actually written is returned unless the write was asynchronous, in which case the number of bytes requested to be transferred is returned. If unsuccessful, a –1 is returned and *errno* is set to indicate the error.

ERRORS

writev will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF] *d* is not a valid descriptor open for writing.

[EPIPE] An attempt is made to write to a pipe that is not open for reading by any process (or to a socket of type SOCK_STREAM that is connected to a peer socket). Note: an attempted write of this kind will also cause you to receive a SIGPIPE signal from the kernel. If you've not made a special provision to catch or ignore this signal, your process will die.

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.

[EFAULT] Part of *iov* or data to be written to the file points outside the process's allocated address space.

[EINTR] The call is forced to terminate prematurely due to the arrival of a signal whose SV_INTERRUPT bit in **sv_flags** is set (see *sigvec*(2) or *signal*(3)).

[EINVAL] The pointer associated with *d* was negative.

[ENOSPC] There is no free space remaining on the file system containing the file.

[EDQUOT] The user's quota of disk blocks on the file system containing the file has been exhausted.

[EIO] An I/O error occurred while reading from or writing to the file system.

[EWOULDBLOCK] The file was marked for non-blocking I/O, and no data could be written immediately.

[EINVAL] The requested transfer size is too big for a single request. In order to prevent possible memory deadlocks, the kernel will refuse to do transfers larger than approximately half the size of physical memory.

[EINVAL] For asynchronous requests, a maximum of one *iov* region may point to a given logical page.

[EINVAL] *iovcnt* was either less than or equal to 0 or greater than 16.

[EINVAL] One of the *iov_len* values in the *iov* array was negative.

[EINVAL] The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.

[EINVAL] Two or more *iov* buffers share the same virtual memory page, and the mode is asynchronous. (This restriction is necessary because the pages are locked in memory during asynchronous transfers.)

[ENODMON] *Migration only:* The file referenced by *d* is under daemon control, but no daemon is present.

EISMIGRATED *Migration only:* A block in the file reference by *d* is migrated, and the migration daemon was unable to stage the file in from secondary storage.

SEE ALSO

asiostat(2), fcntl(2), lseek(2), open(2), pipe(2), select(2), write(2)

NAME

intro – introduction to library functions

DESCRIPTION

This section describes functions that may be found in various libraries. The library functions are those other than the functions which directly invoke ConvexOS system primitives, described in Section 2. This section has the libraries physically grouped together. The functions described in this section are grouped into various libraries:

(3) and (3S)
The straight "3" functions are the standard C library functions. The C library also includes all the functions described in Section 2. The "3S" functions comprise the standard I/O library. Together with the (3X) and (3C) routines, these functions constitute library *libc*, which is automatically loaded by the C compiler *cc*(1). The link editor *ld*(1) searches this library under the "–lc" option. Declarations for some of these functions may be obtained from include files indicated on the appropriate pages.

(3A) This section describes those functions that are in the CONVEX Ada utility library.

(3C) Routines included for compatibility with other systems. In particular, a number of system call interfaces provided in previous releases of 4BSD have been included for source code compatibility. The manual page entry for each compatibility routine indicates the proper interface to use.

(3F) This section describes those functions that are in the FORTRAN utility library. The "3F" functions provide an interface from *fc* programs to the system in the same manner as the C library does for C programs.

(3M) These functions constitute the math library, *libm*.. The link editor searches this library under the "–lm" option. Declarations for these functions may be obtained from the include file *<math.h>*.

(3N) This section describes network library functions that are applicable to the DARPA Internet network.

(3R) These functions comprise the RPC Services Library. This library provides a high-level interface to the RPC Services as well as defining the necessary XDR routines needed to access RPC Services at a much lower level.

(3S) These functions constitute the "standard I/O package", see *intro*(3S). These functions are in the library *libc* already mentioned. Declarations for these functions may be obtained from the include file *<stdio.h>*.

(3V) These functions are part of the COVUE family library routines. The COVUEnet library is the programmer's interface to the Remote File Access System of COVUEnet.

(3X) Various specialized libraries have not been given distinctive captions. Files in which such libraries are found are named on appropriate pages.

FILES

/lib/libc.a
/usr/lib/libm.a
/usr/lib/libc_p.a
/usr/lib/libm_p.a
/usr/lib/libnfars.a

SEE ALSO

intro(3C), intro(3F), intro(3S), intro(3M), intro(3R), intro(3V), intro(3X), nm(1), ld(1), cc(1), intro(2)

DIAGNOSTICS

Functions in the math library (3M) may return conventional values when the function is undefined for the given arguments or when the value is not representable. In these cases the external variable *errno* (see *intro (2)* and *errno.h (3))* is set to the value EDOM (domain error) or ERANGE (range error). The values of EDOM and ERANGE are defined in the include file *<errno.h>*. Many other standard C functions set **errno** to a positive value indicating the error. C programmers may set **errno** to zero, although the C library routines never do.

BACKWARD COMPATIBILITY

The default libraries linked with **cc** command conform to the ANSI and POSIX specifications. The functionality of the routines in the backward-compatible libraries (linked when using the *-pcc* option of **cc**) often is different from the routines in the conforming libraries. When using the *-str* option of the **cc** command, strict conformance to the ANSI standard is supplied. The ANSI standard guarantees the C programmer of a namespace free from conflicting library names. In this mode, the user may use any name not mentioned by ANSI except those beginning with two underscores and those beginning with an underscore and an uppercase letter. As an additional constraint, names beginning with an underscore and a lowercase letter may only be used in a local scope of a routine. If the C programmer uses names that are reserved for the implementation, errors may occur while linking. When using the *-std* option of the **cc** command, additional conformance to the POSIX standard is supplied. In this mode, the ANSI namespace rules are still followed. POSIX does not reserve its names, so the programmer is guaranteed that the accidental redefinition of a POSIX name will not cause any other function to fail.

NAME

intro – introduction to compatibility library functions

DESCRIPTION

These functions constitute the compatibility library portion of *libc*. They are automatically loaded as needed by the C compiler *cc*(1). The link editor searches this library under the "–lc" option. Use of these routines should, for the most part, be avoided. Manual entries for the functions in this library describe the proper routine to use.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
alarm	alarm(3C)	schedule signal after specified time
ftime	time(3C)	get date and time
getpw	getpw(3C)	get name from uid
gtty	stty(3C)	set and get terminal state (defunct)
nice	nice(3C)	set program priority
pause	pause(3C)	stop until signal
rand	rand(3C)	random number generator
signal	signal(3C)	simplified software signal facilities
srand	rand(3C)	random number generator
stty	stty(3C)	set and get terminal state (defunct)
time	time(3C)	get date and time
times	times(3C)	get process times
utime	utime(3C)	set file times
vlimit	vlimit(3C)	control maximum system resource consumption
vtimes	vtimes(3C)	get information about resource utilization

NAME

intro – introduction to mathematical library functions

DESCRIPTION

These functions constitute the C math library, *libm*. The link editor searches this library under the "–lm" option. Declarations for these functions may be obtained from the include file *<math.h>*.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
acos	sin(3M)	double-precision arc cosine
asin	sin(3M)	double-precision arc sine
atan	sin(3M)	double-precision arc tangent
atan2	sin(3M)	double-precision arc tangent with two arguments
cabs	hypot(3M)	double-precision complex absolute value
ceil	floor(3M)	ceiling function
cos	sin(3M)	double-precision cosine
cosh	sinh(3M)	double-precision hyperbolic cosine
exp	exp(3M)	double-precision exponential
fabs	floor(3M)	double-precision absolute value
floor	floor(3M)	floor function
gamma	gamma(3M)	gamma function
hypot	hypot(3M)	double-precision Euclidean distance
ipow	exp(3M)	integer power
j0	j0(3M)	bessel functions
j1	j0(3M)	bessel functions
jn	j0(3M)	bessel functions
log	exp(3M)	double-precision natural logarithm
log10	exp(3M)	double-precision base 10 logarithm
lpow	exp(3M)	long-long-int power
pow	exp(3M)	double-precision power
sacos	sin(3M)	single-precision arc cosine
sasin	sin(3M)	single-precision arc sine
satan	sin(3M)	single-precision arc tangent
satan2	sin(3M)	single-precision arc tangent with two arguments
scabs	hypot(3M)	single-precision complex absolute value
scos	sin(3M)	single-precision cosine
scosh	sinh(3M)	single-precision hyperbolic cosine
sexp	exp(3M)	single-precision exponential
sfabs	floor(3M)	single-precision absolute value
sin	sin(3M)	double-precision sine
sinh	sinh(3M)	double-precision hyperbolic sine
slog	exp(3M)	single-precision natural logarithm
slog10	exp(3M)	single-precision base 10 logarithm
spow	exp(3M)	single-precision power
sqrt	exp(3M)	double-precision square root
tan	sin(3M)	double-precision tangent
tanh	sinh(3M)	double-precision hyperbolic tangent
y0	j0(3M)	bessel functions
y1	j0(3M)	bessel functions
yn	j0(3M)	bessel functions

DIAGNOSTICS

All of the mathematical functions can be called with arguments which lead to error conditions. Errors can be grouped into two categories: domain and range errors. Domain errors occur when

the argument of a function is out of the domain of the function. Range errors occur when the computed value is not representable within the machine's precision, or the size of the argument is such that evaluation of the function would lead to a significant loss of accuracy.

The mathematical error codes are listed in the include file < *errno.h* >. In general, the errors reported are more specific than EDOM (domain error) and ERANGE (range error). For example, the error code MTH_NEG_BASE is returned when the floating-point power function is called with a negative base.

On an error condition, a message is printed to *stderr* and the external variable *errno* (see *intro* (2)) is set to the corresponding error code. The return value is function dependent.

NAME

stdio – standard buffered input/output package

SYNOPSIS

#include <stdio.h>

FILE *stdin;
FILE *stdout;
FILE *stderr;

DESCRIPTION

The functions described in section 3S constitute a user-level buffering scheme. The inline macros *getc* and *putc*(3S) handle characters quickly. The higher level routines *gets*, *fgets*, *scanf*, *fscanf*, *fread*, *puts*, *fputs*, *printf*, *fprintf*, *fwrite* all use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream*, and is declared to be a pointer to a defined type **FILE**. *fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. There are three normally open streams with constant pointers declared in the include file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file

A constant "pointer" **NULL** (0) designates no stream at all.

An integer constant **EOF** (–1) is returned upon end of file or error by integer functions that deal with streams.

Any routine that uses the standard input/output package must include the header file *<stdio.h>* of pertinent macro definitions. The functions and constants mentioned in sections labeled **3S** are declared in the include file and need no further declaration. The constants, and the following "functions" are implemented as macros; redeclaration of these names is perilous: *getc*, *getchar*, *putc*, *putchar*, *feof*, *ferror*, *fileno*.

SEE ALSO

open(2), close(2), read(2), write(2), fread(3S), fseek(3S), f*(3S)

DIAGNOSTICS

The value **EOF** is returned uniformly to indicate that a **FILE** pointer has not been initialized with *fopen*, input (output) has been attempted on an output (input) stream, or a **FILE** pointer designates corrupt or otherwise unintelligible **FILE** data.

For purposes of efficiency, this implementation of the standard library has been changed to line buffer output to a terminal by default and attempts to do this transparently by flushing the output whenever a *read*(2) from the standard input is necessary. This is almost always transparent, but may cause confusion or malfunctioning of programs which use standard I/O routines but use *read*(2) themselves to read from the standard input.

In cases where a large amount of computation is done after printing part of a line on an output terminal, it is necessary to *fflush*(3S) the standard output before going off and computing so that the output will appear.

BUGS

The standard buffered functions do not interact well with certain other library and system functions, especially *vfork* and *abort*.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
clearerr	ferror(3S)	stream status inquiries
fclose	fclose(3S)	close or flush a stream

fdopen	fopen(3S)	open a stream
feof	ferror(3S)	stream status inquiries
ferror	ferror(3S)	stream status inquiries
fflush	fclose(3S)	close or flush a stream
fgetc	getc(3S)	get character or word from stream
fgets	gets(3S)	get a string from a stream
fileno	ferror(3S)	stream status inquiries
fopen	fopen(3S)	open a stream
fprintf	printf(3S)	formatted output conversion
fputc	putc(3S)	put character or word on a stream
fputs	puts(3S)	put a string on a stream
fread	fread(3S)	buffered binary input/output
freopen	fopen(3S)	open a stream
fscanf	scanf(3S)	formatted input conversion
fseek	fseek(3S)	reposition a stream
ftell	fseek(3S)	reposition a stream
fwrite	fread(3S)	buffered binary input/output
getc	getc(3S)	get character or word from stream
getchar	getc(3S)	get character or word from stream
gets	gets(3S)	get a string from a stream
getw	getc(3S)	get character or word from stream
popen	popen(3S)	initiate I/O to/from a process
printf	printf(3S)	formatted output conversion
putc	putc(3S)	put character or word on a stream
putchar	putc(3S)	put character or word on a stream
puts	puts(3S)	put a string on a stream
putw	putc(3S)	put character or word on a stream
rewind	fseek(3S)	reposition a stream
scanf	scanf(3S)	formatted input conversion
setbuf	setbuf(3S)	assign buffering to a stream
setbuffer	setbuf(3S)	assign buffering to a stream
setlinebuf	setbuf(3S)	assign buffering to a stream
sprintf	printf(3S)	formatted output conversion
sscanf	scanf(3S)	formatted input conversion
ungetc	ungetc(3S)	push character back into input stream

NAME

intro – introduction to miscellaneous library functions

DESCRIPTION

These functions constitute minor libraries and other miscellaneous runtime facilities. Most are available only when programming in C. The list below includes libraries which provide device independent plotting functions, terminal independent screen management routines for two dimensional non-bitmap display terminals, functions for managing data bases with inverted indexes, and sundry routines used in executing commands on remote machines. The routines *getdiskbyname*, *rcmd*, *rresvport*, *ruserok*, *getfsent*, *initgroups*, and *rexec* reside in the standard C runtime library "–lc". All other functions are located in separate libraries indicated in each manual entry.

FILES

/lib/libc.a
/usr/lib/libdbm.a
/usr/lib/libtermcap.a
/usr/lib/libcurses.a
/usr/lib/lib2648.a
/usr/lib/libplot.a

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
assert	assert(3X)	program verification
curses	curses(3X)	screen functions with "optimal" cursor motion
dbminit	dbm(3X)	data base subroutines
delete	dbm(3X)	data base subroutines
endfsent	getfsent(3X)	get file system descriptor file entry
fetch	dbm(3X)	data base subroutines
firstkey	dbm(3X)	data base subroutines
getdiskbyname	getdisk(3X)	get disk description by its name
getfsent	getfsent(3X)	get file system descriptor file entry
getfsfile	getfsent(3X)	get file system descriptor file entry
getfsspec	getfsent(3X)	get file system descriptor file entry
getfstype	getfsent(3X)	get file system descriptor file entry
initgroups	initgroups(3X)	initialize group access list
nextkey	dbm(3X)	data base subroutines
plot	plot(3X)	graphics interface
setfsent	getfsent(3X)	get file system descriptor file entry
store	dbm(3X)	data base subroutines
tgetent	termcap(3X)	terminal independent operation routines
tgetflag	termcap(3X)	terminal independent operation routines
tgetnum	termcap(3X)	terminal independent operation routines
tgetstr	termcap(3X)	terminal independent operation routines
tgoto	termcap(3X)	terminal independent operation routines
tputs	termcap(3X)	terminal independent operation routines

NAME

abort – generate a fault

SYNOPSIS

#include <stdlib.h>

void abort(void);

DESCRIPTION

abort sends the signal *SIGABRT* to the executing process. If the process has a handler for *SIGABRT*, the handler is replaced by the default function *SIG_DFL*. Then the old handler of the process is executed. Next, the buffers are flushed and the process is killed with the signal *SIGABRT* and core is dumped.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. *abort* differs in these backward-compatible libraries in the following ways:

- The standard I/O buffers are not flushed in the backward-compatible mode.
- In the backward-compatible library, an illegal instruction is executed, sending **SIGILL** to the process rather than **SIGABRT.**

SEE ALSO

adb(1), sigvec(2), exit(2), kill(2)

DIAGNOSTICS

NAME

abs, labs, div, ldiv – standard integral functions

SYNOPSIS

#include <stdlib.h>

int abs(int j);

long int labs(long int j);

div_t div(int numer, int denom);

ldiv_t ldiv(long int numer, long int denom);

DESCRIPTION

abs and *labs* return the absolute value of the operand.

ldiv and *div* return a structure containing the quotient and remainder of the operands.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- *labs* is unavailable in the backward-compatible mode.
- *div* is unavailable in the backward-compatible mode.
- *ldiv* is unavailable in the backward-compatible mode.

SEE ALSO

floor(3M) for *fabs*

BUGS

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.

NAME

alarm – schedule signal after specified time

SYNOPSIS

unsigned int alarm(seconds)
unsigned int seconds;

DESCRIPTION

alarm() causes signal SIGALRM to be sent to the invoking process in a number of seconds given by the argument. Unless caught or ignored, the signal terminates the process.

Alarm requests are not stacked; successive calls reset the alarm clock. If the argument is 0, any alarm request is canceled. Because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 2147483647 seconds.

RETURN VALUE

The return value is the amount of time previously remaining in the alarm clock.

SEE ALSO

sigpause(2), sigvec(2), signal(3C), sleep(3)

NAME

assert – program verification

SYNOPSIS

#include <assert.h>

void assert(int expression)

DESCRIPTION

Assert is a macro that indicates *expression* is expected to be non-zero. It causes a diagnostic comment on the standard error and then an *abort* when *expression* is false (0). Compilation with **NDEBUG** defined as a macro, causes *assert* to be defined as **((void) 0)**. Multiple inclusions of *<stdio.h>* may result in different definitions of *assert* macro, depending on whether **NDEBUG** has been defined.

DIAGNOSTICS

'Assertion failed: file *f* line *n.*' *F* is the source file and *n* the source line number of the *assert* statement. Core is dumped.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- *assert* in the backward-compatible mode calls *exit* rather than *abort*.
- Multiple inclusion of *<assert.h>* will result in compile time warnings pertaining to macro redefinition in the backwards-compatible mode.

NAME

atof, atoi, atol, atoll – convert ASCII to numbers

SYNOPSIS

#include <stdlib.h>

double atof(const char *nptr);

atoi(const char *nptr);

long int atol(const char *nptr);

long long int atoll(const char *nptr);

DESCRIPTION

These functions convert a string pointed to by *nptr* to floating, integer, and long integer representations, respectively. The first unrecognized character ends the string.

atof recognizes an optional string of spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional 'e' or 'E' followed by an optionally signed integer.

atoi, *atol*, and *atoll* recognize an optional string of spaces, then an optional sign, then a string of digits.

SEE ALSO

scanf(3S), strtod(3)

BUGS

There are no provisions for overflow.

NAME

bcopy, bcmp, bzero, ffs – bit and byte string operations

SYNOPSIS

```
bcopy(b1, b2, length)
char *b1, *b2;
int length;

bcmp(b1, b2, length)
char *b1, *b2;
int length;

bzero(b, length)
char *b;
int length;

ffs(i)
int i;
```

DESCRIPTION

The functions *bcopy*, *bcmp*, and *bzero* operate on variable length strings of bytes. They do not check for null bytes as the routines in *stringcpy*(3), *stringcmp*(3). do.

Bcopy copies *length* bytes from string *b1* to the string *b2*.

Bcmp compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long.

Bzero places *length* 0 bytes in the string *b1*.

Ffs finds the first bit set in the argument passed it and returns the index of that bit. Bits are numbered starting at 1. A return value of 0 indicates the value passed is zero.

BUGS

The *bcmp* and *bcopy* routines take parameters backwards from *strcmp* and *strcpy*.

NAME

cfgetispeed, cfsetispeed, cfgetospeed, cfsetospeed - get/set terminal input/output speed

SYNOPSIS

```
#include <termios.h>

speed_t cfgetospeed(termios_p)
struct termios *termios_p;

int cfsetospeed(termios_p, speed)
struct termios *termios_p;
speed_t speed;

speed_t cfgetispeed(termios_p)
struct termios *termios_p;

int cfsetispeed(termios_p, speed)
struct termios *termios_p;
speed_t speed;
```

DESCRIPTION

cfgetospeed() returns the output baud rate stored in the termios structure pointed to by *termios_p*.

cfsetospeed() sets the output baud rate stored in the termios structure pointed to by *termios_p* to *speed*. The zero baud rate, B0, is used to terminate the connection. If B0 is specified, the modem control lines shall no longer be asserted. Normally, this will disconnect the line.

cfgetispeed() returns the input baud rate stored in the termios structure.

cfsetispeed() sets the input baud rate stored in the termios structure to *speed*. If the input baud rate is set to zero, the input baud rate will be specified by the value of the output baud rate.

The type *speed_t* and the name symbols for the supported speeds are defined in **<termios.h>**.

RETURNS

Both *cfsetispeed()* and *cfsetospeed()* return a value of zero if successful and -1 to indicate the desired speed was not within the range of supported speeds.

NOTES

Attempts to set baud rates unsupported by the hardware are silently ignored.

REFERENCES

tcgetattr(3), tcsetattr(3), tcgetpgrp(3), tcsetpgrp(3).

NAME

clock - get processor time used

SYNOPSIS

#include <time.h>

clock_t clock(void)

DESCRIPTION

The *clock* function determines the processor time used.

RETURN VALUE

The *clock* function returns the implementation's best approximation to the processor time used by the program since the beginning of an implementation-defined era related only to the program invocation. To determine the time in seconds, the value returned by the *clock* function should be divided by the value of the macro **CLOCKS_PER_SEC**. If the processor time used is not available or its value cannot be represented, the function returns the value **(clock_t)-1**.

SEE ALSO

difftime(3), time(3)

NAME

closeshares - close share data base file

SYNOPSIS

int closeshares()

DESCRIPTION

closeshares closes the share data base file (if open) which otherwise remains open across share file routine calls.

FILES

/etc/shares	share data-base file
/usr/lib/libshare.a	the share-specific library routines

SEE ALSO

getshares(3), getshput(3), openshares(3), putshares(3), sharesfile(3)

DIAGNOSTICS

closeshares always returns 0.

NOTE

The Share scheduler is an optional product; for more information, contact your CONVEX sales representative.

NAME

crypt, setkey, encrypt – encryption operations

SYNOPSIS — UNITED STATES DISTRIBUTION

```
char *crypt(key, salt)
char *key, *salt;

setkey(key)
char *key;

encrypt(block, edflag)
char *block;
```

SYNOPSIS — INTERNATIONAL DISTRIBUTION

```
char *crypt(key, salt)
char *key, *salt;

encrypt(block)
char *block;
```

DESCRIPTION — UNITED STATES DISTRIBUTION

Crypt is the password encryption routine. It is based on the NBS Data Encryption Standard, with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

The first argument to *crypt* is normally a user's typed password. The second is a 2-character string chosen from the set [a-zA-Z0-9./]. The *salt* string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password, in the same alphabet as the salt. The first two characters are the salt itself.

The other entries provide (rather primitive) access to the actual DES algorithm. The argument of *setkey* is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored, leading to a 56-bit key which is set into the machine.

The argument to the *encrypt* entry is likewise a character array of length 64 containing 0's and 1's. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by *setkey*. If *edflag* is 0, the argument is encrypted; if nonzero, it is decrypted.

DESCRIPTION — INTERNATIONAL DISTRIBUTION

Crypt is the password encryption routine. It is based on a one-way hashing encryption algorithm with variations intended (among other things) to frustrate use of hardware implementations of a key search.

The first argument to *crypt* is normally a user's typed password. The second is a 2-character string chosen from the set [a-zA-Z0-9./]. The *salt* string is used to perturb the hashing algorithm in 1 of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password, in the same alphabet as the salt. The first two characters are the salt itself.

There is a character array of length 64 containing only the characters with numerical value 0 and 1. When this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine by crypt.

The *encrypt* entry provides (rather primitive) access to the actual hashing algorithm. The argument to the *encrypt* entry is a character array of length 64 containing only the characters with numerical value of 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by *crypt*.

SEE ALSO

passwd(1), passwd(5), login(1), getpass(3)

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

ctermid – generate terminal pathname

SYNOPSIS

#include <stdio.h>

char *ctermid(s)
char *s;

DESCRIPTION

The *ctermid()* function generates a string that, when used as a pathname, refers to the current controlling terminal for the current process.

If the *ctermid()* function returns a pathname, access to the file is not guaranteed.

RETURNS

If **s** is a **NULL** pointer, the string is generated in an area that may be static (and therefore may be overwritten by each call), the address of which is returned. Otherwise *s* is assumed to point to a character array of at least L_ctermid bytes; the string is placed in this array and the value of *s* is returned. The symbolic constant L_ctermid is defined in **<stdio.h>**, and shall have a value greater than zero.

The *ctermid()* function returns an empty string if the pathname that would refer to the controlling terminal cannot be determined.

SEE ALSO

ttyname(3)

NAME

ctime, localtime, gmtime, asctime, timezone, mktime – date and time manipulation routines

SYNOPSIS

```
#include <time.h>

char *ctime(const time_t *clock);

struct tm *localtime(const time_t *clock);

struct tm *gmtime(const time_t *clock);

char *asctime(const time_t *tm);

char *timezone(int zone, int dst);

time_t mktime(struct tm *timeptr);
```

DESCRIPTION

ctime converts a time pointed to by *clock* such as returned by *time*(3C) into ASCII and returns a pointer to a 26-character string in the following form. (All the fields have constant width.)

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

localtime and *gmtime* return pointers to structures containing the broken-down time. *localtime* corrects for the time zone and possible daylight savings time; *gmtime* converts directly to GMT, which is the time UNIX uses. *asctime* converts a broken-down time to ASCII and returns a pointer to a 26-character string.

The structure declaration from the include file is:

```
struct tm {
        int tm_sec;     /* 0-59    seconds */
        int tm_min;     /* 0-59    minutes */
        int tm_hour;    /* 0-23    hour */
        int tm_mday;    /* 1-31    day of month */
        int tm_mon;     /* 0-11    month */
        int tm_year;    /* 0-      year - 1900 */
        int tm_wday;    /* 0-6     day of week (Sunday = 0) */
        int tm_yday;    /* 0-365   day of year */
        int tm_isdst;   /* flag:   daylight savings time in effect */
};
```

These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday = 0), year – 1900, day of year (0-365), and a flag that is nonzero if daylight saving time is in effect.

When local time is called for, the program consults the system to determine the time zone and whether the U.S.A., Australian, Eastern European, Middle European, or Western European daylight saving time adjustment is appropriate. The program knows about various peculiarities in time conversion over the past 10-20 years; if necessary, this understanding can be extended.

timezone returns the name of the time zone associated with its first argument, which is measured in minutes westward from Greenwich. If the second argument is 0, the standard name is used, otherwise the Daylight Saving version. If the required name does not appear in a table built into the routine, the difference from GMT is produced; e.g., in Afghanistan *timezone(-(60*4+30), 0)* is appropriate because it is 4:30 ahead of GMT and the string **GMT+4:30** is produced.

mktime converts the broken down local time, contained in a *struct tm* into a calendar time value with the same encoding as that of the values returned by the *time* function. The original values of the *tm_wday* and *tm_yday* components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated. This allows positive values of *tm_isdst* to indicate that Daylight Saving Time is initially in effect, zero to indicate that Daylight

Saving Time is not initially in effect, and negative values indicate that *mktime* is to attempt to determine if Daylight Saving Time is in effect for the specified time.

Upon successful completion, the values of the *tm_wday* and *tm_yday* components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above; the final value of *tm_mday* is not set until *tm_mon* and *tm_year* are determined.

SEE ALSO

gettimeofday(2), time(3c), tzset(3)

BUGS

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

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii, toupper, tolower, _toupper, _tolower, toascii – character testing and mapping macros

SYNOPSIS

#include <ctype.h>

int isalpha(int c);

. . .

DESCRIPTION

These functions and macros are used for testing and mapping characters.

The following character testing macros classify ASCII-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. *isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value EOF (see *stdio*(3S)).

isalpha	*c* is a letter
isupper	*c* is an uppercase letter
islower	*c* is a lowercase letter
isdigit	*c* is a digit
isxdigit	*c* is a hexadecimal digit
isalnum	*c* is an alphanumeric character
isspace	*c* is a space, tab, carriage return, newline, vertical tab, or formfeed
ispunct	*c* is a punctuation character (neither control nor alphanumeric)
isprint	*c* is a printing character, code 040(8) (space) through 0176 (tilde)
isgraph	*c* is a printing character, similar to *isprint* except false for space
iscntrl	*c* is a delete character (0177) or ordinary control character (less than 040)
isascii	*c* is an ASCII character, code less than 0200

The next five macros and functions are used for character mapping. The first two are functions; the remaining are macros.

tolower	if the argument is an uppercase letter, the corresponding lowercase letter is returned; otherwise the argument is returned unchanged.
toupper	if the argument is a lowercase letter, the corresponding uppercase letter is returned; otherwise the argument is returned unchanged.
_tolower	accomplishes the same thing as *tolower,* but has a restricted domain and is faster. If the argument to *_tolower* is not an upper-case letter, its result is undefined.
_toupper	accomplishes the same thing as *toupper,* but has a restricted domain and is faster. If the argument to *_toupper* is not an lower-case letter, its result is undefined.
toascii	yields its argument with all bits turned off that are not part of the standard ASCII character.

SEE ALSO

ascii(7)

BUGS

toupper and *tolower* are System V compatible and if you intend to write codes that will be ported

to other BSD systems you should use the following constructs:

```
if(isupper(c))
    c = tolower(c);

if(islower(c))
    c = toupper(c);
```

Additionally, if speed is an issue, you could include the following defines in your CONVEX source.

```
#define tolower _tolower
#define toupper _toupper
```

NAME

curses – screen functions with "optimal" cursor motion

SYNOPSIS

#include <curses.h>

cc [flags] files **–lcurses –ltermcap** [libraries]

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the *refresh()* tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine *initscr()* must be called before any of the other routines that deal with windows and screens are used. The routine *endwin()* should be called before exiting.

SEE ALSO

"Screen Updating and Cursor Movement Optimization: A Library Package" in the *CONVEX UNIX Tutorial Papers*
ioctl(2), getenv(3), tty(4), termcap(5)

AUTHOR

Ken Arnold

FUNCTIONS

addch(ch)	add a character to *stdscr*
addstr(str)	add a string to *stdscr*
box(win,vert,hor)	draw a box around a window
crmode()	set *cbreak* mode
clear()	clear *stdscr*
clearok(scr,boolf)	set clear flag for *scr*
clrtobot()	clear to bottom on *stdscr*
clrtoeol()	clear to end of line on *stdscr*
delch()	delete a character
deleteln()	delete a line
delwin(win)	delete *win*
echo()	set *echo* mode
endwin()	end *window* modes
erase()	erase *stdscr*
getch()	get a character through *stdscr*
getcap(name)	get terminal capability *name*
getstr(str)	get a string through *stdscr*
gettmode()	get *tty* modes
getyx(win,y,x)	get (y,x) coordinates
inch()	get character at current (y,x) coordinates
initscr()	initialize screens
insch(c)	insert a character
insertln()	insert a line
leaveok(win,boolf)	set leave flag for *win*
longname(termbuf,name)	get long name from *termbuf*
move(y,x)	move to (y,x) on *stdscr*
mvcur(lasty,lastx,newy,newx)	actually move cursor
newwin(lines,cols,begin_y,begin_x)	create a new window
nl()	set newline mapping
nocrmode()	unset *cbreak* mode
noecho()	unset *echo* mode
nonl()	unset newline mapping

noraw()	unset *raw* mode
overlay(win1,win2)	overlay win1 on win2
overwrite(win1,win2)	overwrite win1 on top of win2
printw(fmt,arg1,arg2,...)	printf on *stdscr*
raw()	set *raw* mode
refresh()	make current screen look like *stdscr*
resetty()	reset *tty* flags to stored value
savetty()	stored current *tty* flags
scanw(fmt,arg1,arg2,...)	*scanf* through *stdscr*
scroll(win)	scroll *win* one line
scrollok(win,boolf)	set scroll flag
setterm(name)	set term variables for name
standend()	end *standout* mode
standout()	start *standout* mode
subwin(win,lines,cols,begin_y,begin_x)	create a subwindow
touchwin(win)	"change" all of *win*
unctrl(ch)	printable version of *ch*
waddch(win,ch)	add character to *win*
waddstr(win,str)	add string to *win*
wclear(win)	clear *win*
wclrtobot(win)	clear to bottom of *win*
wclrtoeol(win)	clear to end of line on *win*
wdelch(win,c)	delete character from *win*
wdeleteln(win)	delete line from *win*
werase(win)	erase *win*
wgetch(win)	get a character through *win*
wgetstr(win,str)	get a string through *win*
winch(win)	get character at current (y,x) in *win*
winsch(win,c)	insert character into *win*
winsertln(win)	insert line into *win*
wmove(win,y,x)	set current (y,x) coordinates on *win*
wprintw(win,fmt,arg1,arg2,...)	*printf* on *win*
wrefresh(win)	make screen look like *win*
wscanw(win,fmt,arg1,arg2,...)	*scanf* through *win*
wstandend(win)	end *standout* mode on *win*
wstandout(win)	start *standout* mode on *win*

NAME

cuserid, getlogin – get user name

SYNOPSIS

char *getlogin()

#include <stdio.h>

char *cuserid(s)
char *s;

DESCRIPTION

These functions return a string giving the name of the user associated with the current process. The *cuserid()* function returns a name associated with the effective user ID of the process, and the *getlogin()* function returns the name associated by the login activity with the control terminal.

The recommended procedure is either to call the *cuserid()* function, or to call *getlogin()* and, if that fails, to call the *getpwuid()* function with the name returned by the *getuid()* function.

The *getlogin()* function returns a pointer to the user's login name. The same user ID may be shared by several login names. Therefore, to ensure that the correct user database entry is found, the *getlogin()* function should be used with the *getpwnam()* function.

If *getlogin()* returns a non-**NULL** pointer, that pointer is to the name the user logged in under, even if there are several login names with the same user ID.

The *cuserid()* function generates a character representation of the login name of the owner of the current process. If **s** is not a **NULL** pointer, it is assumed that **s** points to an array of at least L_cuserid bytes; the representation is returned in this array. The symbolic constant L_cuserid is defined in **<stdio.h>**, and shall have a value greater than zero.

RETURNS

The *getlogin()* function returns a pointer to a string containing the user's login name, or a **NULL** pointer if the user's login name cannot be found.

The return value from *getlogin()* may point to static data and therefore may be overwritten by each call.

If **s** is a **NULL** pointer, the result from *cuserid()* is generated in an area that may be static, the address of which is returned. If the login name cannot be found, *cuserid()* returns a **NULL** Pointer. If **s** is not a **NULL** pointer, **s** is returned. If the login name cannot be found, the null character shall be placed at ***s**. The return value from *cuserid() may point to static data* overwritten by each call.

The implementation of the *cuserid()* function may use the *getpwnam()* function, as the results of a user's call to either routine may be overwritten by a subsequent call to the other routine.

SEE ALSO

getpwnam(3), getpwuid(3)

NAME

dbminit, fetch, store, delete, firstkey, nextkey – data base subroutines

SYNOPSIS

```
#include <dbm.h>

typedef struct {
        char *dptr;
        int dsize;
} datum;

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;
```

DESCRIPTION

Note: the dbm library has been superceded by ndbm(3), and is now implemented using ndbm. These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. The functions are obtained with the loader option **–ldbm**.

Keys and *contents* are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has '.dir' as its suffix. The second file contains all data and has '.pag' as its suffix.

Before a database can be accessed, it must be opened by *dbminit*. At the time of this call, the files *file*.**dir** and *file*.**pag** must exist. (An empty database is created by creating zero-length '.dir' and '.pag' files.)

Once open, the data stored under a key is accessed by *fetch* and data is placed under a key by *store*. A key (and its associated contents) is deleted by *delete*. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of *firstkey* and *nextkey*. *Firstkey* will return the first key in the database. With any key *nextkey* will return the next key in the database. This code will traverse the data base:

```
for (key = firstkey(); key.dptr != NULL; key = nextkey(key))
```

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates ok. Routines that return a *datum* indicate errors with a null (0) *dptr*.

SEE ALSO

ndbm(3)

BUGS

The '.pag' file will contain holes so that its apparent size is about four times its actual content. Older UNIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (cp, cat, tp, tar, ar) without filling in the holes.

Dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover all key/content pairs that hash together must fit on a single block. *Store* will return an error in the event that a disk block fills with inseparable data.

Delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by *firstkey* and *nextkey* depends on a hashing function, not on anything interesting.

NAME

difftime - compute the difference between two times.

SYNOPSIS

#include <time.h>

double difftime(time_t time1, time_t time0);

DESCRIPTION

difftime computes the difference between two calendar times: **time1 - time0.**

RETURN VALUE

difftime returns the difference expressed in seconds as a **double**.

SEE ALSO

time(3c)

NAME

opendir, readdir, rewinddir, closedir – directory operations

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(filename)
char *filename;

struct dirent *readdir(dirp)
DIR *dirp;

rewinddir(dirp)
DIR *dirp;

closedir(dirp)
DIR *dirp;
```

DESCRIPTION

opendir() opens the directory named by *filename* 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 *filename* cannot be accessed, or if it cannot *malloc*(3) enough memory to hold the whole thing.

readdir() returns a pointer to the next directory entry. It returns **NULL** upon reaching the end of the directory (or detecting an invalid *seekdir()* operation, see **CONVEX EXTENSIONS** below). *rewinddir()* resets the position of the named *directory stream* to the beginning of the directory.

closedir() closes the named *directory stream* and frees the structure associated with the DIR pointer.

Sample code that searches a directory for entry "name" is:

```
len = strlen(name);
dirp = opendir(".");
for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp))
        if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
                closedir(dirp);
                return FOUND;
        }
closedir(dirp);
return NOT_FOUND;
```

CONVEX EXTENSIONS

```
long telldir(dirp)
DIR *dirp;

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

seekdir() sets the position of the next *readdir()* operation on the *directory stream.* The new position reverts to the one associated with the *directory stream* when the *telldir()* operation was performed. Values returned by *telldir()* are good only for the lifetime of the DIR pointer from which they are derived. If the directory is closed and then reopened, the *telldir()* value may be invalidated due to undetected directory compaction. It is safe to use a previous *telldir()* value

immediately after a call to *opendir()* and before any calls to *readdir()*.

BACKWARD COMPATIBILITY

Old versions of the operating system wanted **<sys/dir.h>**, not **<dirent.h>**, to be included.

SEE ALSO

open(2), close(2), getdirentries(2), lseek(2), read(2), getwd(3), scandir(3), dir(5)

NAME

ecvt, fcvt, gcvt – output conversion

SYNOPSIS

```
char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt(value, ndigit, buf)
double value;
char *buf;
```

DESCRIPTION

Ecvt converts the *value* to a null-terminated string of *ndigit* ASCII digits and returns a pointer thereto. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero. The low-order digit is rounded.

Fcvt is identical to *ecvt*, except that the correct digit has been rounded for Fortran F-format output of the number of digits specified by *ndigits*.

Gcvt converts the *value* to a null-terminated ASCII string in *buf* and returns a pointer to *buf*. It attempts to produce *ndigit* significant digits in Fortran F format if possible, otherwise E format, ready for printing. Trailing zeros may be suppressed.

SEE ALSO

printf(3s)

BUGS

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

NAME

end, etext, edata – last locations in program

SYNOPSIS

extern end;
extern etext;
extern edata;

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of *etext* is the first address above the program text, *edata* above the initialized data region, and *end* above the uninitialized data region.

When execution begins, the program break coincides with *end,* but it is reset by the routines *brk*(2), *malloc*(3), standard input/output (*stdio*(3)), the profile (**–p**) option of *cc*(1), etc. The current value of the program break is reliably returned by 'sbrk(0)', see *brk*(2).

SEE ALSO

brk(2), malloc(3)

NAME

erf, erfc – error functions

SYNOPSIS

```
#include <math.h>

double erf(x)
double x;

double erfc(x)
double x;
```

DESCRIPTION

Erf(x) returns the error function of x; where erf(x) := $(2/\sqrt{\pi})\int_0^x \exp(-t^2)\,dt$.

Erfc(x) returns 1.0–erf(x).

The entry for erfc is provided because of the extreme loss of relative accuracy if erf(x) is called for large x and the result subtracted from 1. (e.g. for x = 10, 12 places are lost).

SEE ALSO

math(3M)

NAME

errno.h, errno – header file relating to error reporting

SYNOPSIS

#include <errno.h>

int errno;

DESCRIPTION

The value of **errno** is zero at program startup. Library functions set **errno** to positive values indicating various error conditions. No library function sets **errno** to zero, although the C programmer may do so. The programmer should not set **errno** to any value except zero. *errno.h* contains the values (macros) that **errno** may acquire. All names beginning with **E** and followed by a digit or uppercase letter are reserved by the implementation for use as macro definitions for error conditions.

NAME

execl, execle, execlp, execv, execvp, environ – execute a file

SYNOPSIS

```
execl(name, arg0, arg1, ..., argn, (char *)0)
char *name, *arg0, *arg1, ..., *argn;

execle(name, arg0, arg1, ..., argn, (char *)0, envp)
char *name, *arg0, *arg1, ..., *argn, *envp[];

execv(name, argv)
char *name, *argv[];

extern char **environ;
```

DESCRIPTION

These routines provide various interfaces to the *execve*(2) system call. Refer to *execve*(2) for a description of their properties; only brief descriptions are provided here.

Exec in all its forms overlays the calling process with the named file, then transfers to the entry point of the core image of the file. There can be no return from a successful exec; the calling core image is lost.

The *name* argument is a pointer to the name of the file to be executed. The pointers *arg*[*0*], *arg*[*1*] ... address null-terminated strings. Conventionally *arg*[*0*] is the name of the file.

Two interfaces are available. *execl()* is useful when a known file with known arguments is being called; the arguments to *execl()* are the character strings constituting the file and the arguments; the first argument is conventionally the same as the file name (or its last component). A null pointer argument must end the argument list.

The *execv()* version is useful when the number of arguments is unknown in advance; the arguments to *execv()* are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a null pointer.

When a C program is executed, it is called as follows:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least 1 and the first member of the array points to a string containing the name of the file.

argv is directly usable in another *execv()* because *argv*[*argc*] is 0.

envp is a pointer to an array of strings that constitute the *environment* of the process. Each string consists of a name, an "=", and a null-terminated value. The array of pointers is terminated by a null pointer. The shell *sh*(1) passes an environment entry for each global shell variable defined when the program is called. See *environ*(7) for some conventionally used names. The C run-time start-off routine places a copy of *envp* in the global cell *environ*, which is used by *execv()* and *execl()* to pass the environment to any subprograms executed by the current program.

execlp() and *execvp()* are called with the same arguments as *execl()* and *execv()*, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

RETURN VALUES

The *execl* calls return a -1 if the file cannot be found, or if it is not executable, or if it does not start with a valid magic number (see *a.out*(5)), or if maximum memory is exceeded, or if the arguments require too much space,

There is no return from a successful call.

CONVEX EXTENSIONS

exect(name, argv, envp)
char *name, *argv[], *envp[];

The *exect()* version is used when the executed file is to be manipulated with *pattach*(2). The program is forced to single step a single instruction giving the parent an opportunity to manipulate its state.

NOTES

If *execvp()* is called to execute a file that turns out to be a shell command file, and if it is impossible to execute the shell, the values of *argv*[*0*] and *argv*[*-1*] will be modified before return.

execvp() should know that files beginning with a "#" are intended to be processed by /bin/csh and not /bin/sh.

At least 1 of the execute-permission bits must be set for a file to be executed, even for the superuser.

SEE ALSO

sh(1), csh(1), execve(2), fork(2), environ(7)

NAME

atexit, exit – terminate a process after flushing any pending output

SYNOPSIS

```
#include <stdlib.h>

int atexit(void (*func)(void));

int exit(int status);
```

DESCRIPTION

atexit registers the function, *func*, for execution at program termination; that is, *func* will be executed when the program exits. On program exit, functions registered are executed in the reverse order of registration.

exit terminates a process after flushing any buffered output.

RETURNS

atexit returns 0 if successful and non-zero if the request could not be honored. Currently, up to 50 functions may be registered with the *atexit* function.

exit never returns.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- *atexit* is unavailable in the backward-compatible mode.

SEE ALSO

exit(2), intro(3S)

NAME

exp, ipow, log, log10, lpow, pow, sqrt, expf, logf, log10f, powf, sqrtf, sexp, slog, slog10, spow, ssqrt – exponential, logarithm, power, square root

SYNOPSIS

```
#include <math.h>

double exp(double x);

int ipow(int x, int y);

double log(double x);

double log10(double x);

long long int lpow(long long int

double pow(double x, double y);

double sqrt(double x);

float expf(float x);

float logf(float x);

float log10f(float x);

float powf(float x, float y);

float sqrtf(float x);
```

DESCRIPTION

exp returns the double-precision exponential function of x; *expf* returns the single-precision exponential.

log returns the double-precision natural logarithm of x; *logf* returns the single-precision natural logarithm.

log10 returns the double-precision base 10 logarithm of x; *log10f* returns the single-precision base 10 logarithm.

pow returns x^y for double-precision x and y; *powf* returns x^y for single-precision x and y. *ipow* returns x^y for integer x and y; *lpow* returns x^y for long long integer x and y.

sqrt returns the double-precision square root of x; *sqrtf* returns the single-precision square root.

SEE ALSO

hypot(3M), sinh(3M), intro(3M)

DIAGNOSTICS

The exponential, logarithmic, and power functions set **errno** to **EDOM** on domain error. The power and exponential functions set **errno** to **ERANGE** on range error. On overflow, **HUGE_VAL** of the same sign is returned.

Evaluation of the exponential function can lead to overflow. The valid range of arguments is [**ln**(*xmin*),**ln**(*xmax*)], where *xmin* is the smallest positive floating-point value and *xmax* is the largest positive floating-point value. Arguments outside of this range are replaced by **ln**(*xmin*) or **ln**(*xmax*) and evaluation is continued. The logarithm functions are undefined for nonpositive arguments. If x is negative, evaluation continues with $|x|$. If x is zero, the most negative floating-point value is returned. The power functions are undefined when x in x^y is negative and y is not integral. In this case, evaluation continues with $|x|$. The power functions are undefined when y in x^y is nonpositive, and x is zero. In this case, the largest floating-point value is returned. The square root functions are undefined for negative x. The square root functions return **sqrt**(**abs**(x)) for negative x.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- The names of the floating-point routines are *sexp, slog, slog10, spow,* and *ssqrt.*
- In the backward-compatible mode **errno** is set to **MTH_OVF_EXP, MTH_UNDEF_LOG, MTH_NEG_BASE, MTH_ZERO_BASE, MTH_OVF_POW,** or **MTH_UNDEF_SQRT** on domain and range error. A message describing the error is printed.

NAME

exportent, getexportent, setexportent, addexportent, remexportent, endexportent, getexportopt – get exported file system information

SYNOPSIS

```
#include <stdio.h>
#include <exportent.h>
FILE *setexportent()
struct exportent *getexportent(filep)
        FILE *filep;
int addexportent(filep, dirname, options)
        FILE *filep;
        char *dirname;
        char *options;
int remexportent(filep, dirname)
        FILE *filep;
        char *dirname;
char *getexportopt(xent, opt)
        struct exportent *xent;
        char *opt;
void endexportent(filep)
        FILE *filep;
```

DESCRIPTION

These routines access the exported filesystem information in */etc/xtab*.

setexportent() opens the export information file and returns a file pointer to use with *getexportent*, *addexportent*, *remexportent*, and *endexportent*. **getexportent()** reads the next line from *filep* and returns a pointer to an object with the following structure containing the broken-out fields of a line in the file, */etc/xtab* The fields have meanings described in *exports*(5).

```
#define ACCESS_OPT  "access"  /* machines that can mount fs */
#define ROOT_OPT    "root"    /* machines with root access of fs */
#define RO_OPT      "ro"      /* export read-only */
#define RW_OPT      "rw"      /* export read-mostly */
#define ANON_OPT    "anon"    /* uid for anonymous requests */
#define ASYNC_OPT   "async"   /* export for asynchronous writes */
struct exportent {
        char *xent_dirname;  /* directory (or file) to export */
        char *xent_options;  /* options, as above */
};
```

addexportent() adds the *exportent()* to the end of the open file *filep*. It returns 0 if successful and –1 on failure. *remexportent()* removes the indicated entry from the list. It also returns 0 on success and –1 on failure. *getexportopt()* scans the *xent_options* field of the *exportent()* structure for a substring that matches *opt*. It returns the string value of *opt*, or NULL if the option is not found.

endexportent() closes the file.

FILES

/etc/exports
/etc/xtab

SEE ALSO

exports(5), exportfs(8)

DIAGNOSTICS

NULL pointer (0) returned on EOF or error.

BUGS

The returned *exportent()* structure points to static information that is overwritten in each call.

NOTES

exportent is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

#include <stdio.h>

int fclose(FILE *stream);

int fflush(FILE *stream);

DESCRIPTION

fclose causes any buffers for the named *stream* to be emptied, and the file to be closed. Buffers allocated by the standard input/output system are freed.

fclose is performed automatically upon calling *exit*(3).

fflush causes any buffered data for the named output *stream* to be written to that file. If the file is opened for reading, unread buffered data is invalidated. The stream remains open. If **NULL** is supplied as an argument, all the open streams are flushed.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- The backward-compatible mode does not recognize **NULL** as a special argument to *fflush*.
- The backward-compatible mode does not always set **errno** as the conforming *fflush* does.
- *fflush* in backward-compatible mode does not invalidate the buffer or call the *lseek* routine.
- *fclose* calls *fflush* to flush the buffers. *fclose* differs as *fflush* differs in the backward-compatible mode.

SEE ALSO

close(2), fopen(3S), lseek(2), setbuf(3S)

DIAGNOSTICS

These routines return **EOF** if *stream* is not associated with an output file, or if buffered data cannot be transferred to that file.

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS

```
#include <stdio.h>
int feof(FILE *stream);
int ferror(FILE *stream);
void clearerr(FILE *stream);
int fileno(FILE *stream);
```

DESCRIPTION

feof returns non-zero when end of file is read on the named input *stream*, otherwise zero.

ferror returns non-zero when an error has occurred reading or writing the named *stream*, otherwise zero. Unless cleared by *clearerr*, the error indication lasts until the stream is closed.

clearerr resets the error indication on the named *stream*.

fileno returns the integer file descriptor associated with the *stream*, see *open*(2).

These functions are implemented as macros; they cannot be redeclared.

SEE ALSO

fopen(3S), open(2)

NAME

float.h – header file containing information on floating-point numbers

SYNOPSIS

#include <float.h>

DESCRIPTION

float.h contains macros for describing the floating-point representation available on the machine.

FLT_RADIX is the radix of the machine. Because CONVEX uses a binary representation the **FLT_RADIX** is 2.

In the following macros, a suffix of **FLT** refers to a **float; DBL** refers to a **double;** and **LDBL** refers to a **long double.**

FLT_MANT_DIG, DBL_MANT_DIG, and **LDBL_MANT_DIG** describe the number bits (binary digits) in the mantissa.

FLT_DIG, DBL_DIG, and **LDBL_DIG** are the number of accurate decimal digits a floating-point number.

FLT_MIN_EXP, DBL_MIN_EXP, and **LDBL_MIN_EXP** are the smallest negative integers such that 2 to that power minus one are normalized floating-point numbers.

FLT_MIN_10_EXP, DBL_MIN_10_EXP, and **LDBL_MIN_10_EXP** are the smallest negative integers such that 10 raised to that power are in the range of normalized floating-point numbers.

FLT_MAX_EXP, DBL_MAX_EXP, and **LDBL_MAX_EXP** are the largest integers such that 2 to that power minus one are normalized floating-point numbers.

FLT_MIN_10_EXP, DBL_MIN_10_EXP, and **LDBL_MIN_10_EXP** are the largest integers such that 10 raised to that power are in the range of normalized floating-point numbers.

FLT_MAX, DBL_MAX, LDBL_MAX, FLT_MIN, DBL_MIN, and **LDBL_MIN** are the largest and smallest floating-point numbers.

FLT_EPSILON, DBL_EPSILON, and **LDBL_EPSILON** represent the difference between 1.0 and the next representable floating-point number.

NAME

fabs, floor, ceil, fabsf, sfabs – absolute value, floor, ceiling functions

SYNOPSIS

#include <math.h>

double floor(double x);

double ceil(double x);

double fabs(double x);

float fabsf(float x);

DESCRIPTION

fabs returns the double-precision absolute value $|x|$; *fabsf* returns the single-precision absolute value $|x|$.

floor returns the largest integer not greater than x.

ceil returns the smallest integer not less than x.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- The name of the floating-point routine is *sfabs*.

SEE ALSO

abs(3)

NAME

fopen, freopen, fdopen – open a stream

SYNOPSIS

#include <stdio.h>

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

FILE *freopen(const char *filename, const char *mode);

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

DESCRIPTION

fopen opens the file named by *filename* and associates a stream with it. *fopen* returns a pointer to be used to identify the stream in subsequent operations.

mode is a character string having one of the following values:

"r" open for reading

"w" create (or truncate to zero length) for writing

"a" append: open for writing at end of file, or create for writing. All subsequent writes go to the end of the file regardless of intervening calls to *fseek* or other file positioning functions. The file position indicator is placed at the end of the file when the file is opened.

"rb", "wb", and "ab" are the same as "r", "w", and "a". They indicate binary files and are for compatibility with other operating systems because text files and binary files are identical on ConvexOS.

"r+" open for reading and writing.

"w+" create (or truncate to zero length) for writing and reading.

"a+" Open for appending and reading. All subsequent writes go to the end of the file regardless of intervening calls to the *fseek* or other file positioning functions. The file position indicator is placed at the end of the file when the file is opened.

"rb+","r+b","wb+","w+b","ab+", or "a+b" are the same as "r+","w+", and "a+". They indicate binary files and are for compatibility with other operating systems because text files and binary files are identical on ConvexOS.

Both reads and writes may be used on read/write streams, with the limitation that an *fseek, rewind,* or reading an end-of-file must be used between a read and a write or vice-versa.

freopen substitutes the named file in place of the open *stream*. It returns the original value of *stream*. The original stream is closed.

freopen is typically used to attach the preopened constant names, **stdin, stdout, stderr,** to specified files.

fdopen associates a stream with a file descriptor obtained from *open, dup, creat,* or *pipe*(2). The *mode* of the stream must agree with the *mode* of the open file.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- Files opened for append with the backward-compatible *fopen* and *freopen* do not insure that all writes are to the end of the file. Writes occur at the location specified by the *current* file position indicator. The initial position of the file position indicator is to the end of the file.
- The binary modes are not available with *fopen* and *freopen* in the backward-compatible

libraries.

SEE ALSO

open(2), fclose(3s), fseek(3s)

DIAGNOSTICS

fopen and *freopen* return **NULL** if *filename* cannot be accessed.

NAME

fread, fwrite – buffered binary input/output

SYNOPSIS

#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

DESCRIPTION

fread reads, into a block beginning at *ptr*, *nmemb* items of data of size *size* from the named input *stream*. It returns the number of items actually read. If *stream* is **stdin** and the standard output is line buffered, then any partial output line will be flushed before any call to *read*(2) to satisfy the *fread*.

fwrite appends at most *nmemb* items of data of size *size* beginning at *ptr* , a pointer to the named output *stream*. It returns the number of items actually written.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- **Errno** is not always set on error in the backward compatible mode.

SEE ALSO

read(2), write(2), fopen(3S), getc(3S), putc(3S), gets(3S), puts(3S), printf(3S), scanf(3S)

DIAGNOSTICS

fread and *fwrite* return 0 upon end of file or error and **errno** is set to a nonzero value.

NAME

frexp, ldexp, modf, fmod – split into mantissa and exponent

SYNOPSIS

```
#include <math.h>

double frexp(double value, int *eptr);

double ldexp(double value, int exp);

double modf(double value, double *iptr);

double fmod(double x, double y);
```

DESCRIPTION

frexp returns the mantissa of a double *value* as a double quantity, x, of magnitude less than 1 and stores an integer n such that $value = x * 2^n$ indirectly through *eptr*.

ldexp returns the quantity $value * 2^{exp}$.

modf returns the positive fractional part of *value* and stores the integer part indirectly through *iptr*.

fmod computes the floating point remainder of x and y.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- *fmod* is not available in the backward-compatible mode.

NAME

fseek, ftell, fsetpos, fgetpos, rewind – reposition a stream

SYNOPSIS

#include <stdio.h>

int fseek(FILE *stream, long int offset,

long int ftell(FILE *stream);

int fsetpos(FILE *stream, const fpos_t *pos);

int fgetpos(FILE *stream, fpos_t *pos);

void rewind(FILE *stream);

DESCRIPTION

fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, according as *ptrname* has the value **SEEK_SET** (beginning of file), **SEEK_CUR** (current value of file position indicator), or **SEEK_END** (end of file). *fseek* undoes any effects of *ungetc*(3S).

ftell returns the current value of the offset relative to the beginning of the file associated with the named *stream*. It is measured in bytes on ConvexOS.

On error, -1 is returned and **errno** is set to the values: **EBADF, ESPIPE,** or **EINVAL.**

fsetpos sets the file pointer according to the value of the object to pointed to by *pos*. This value may only be obtained by an earlier call to *fgetpos* on the same stream. The *eof* indicator will be cleared for that file. Input or output may be followed by a call to *fgetpos*. *fgetpos* gets the current file position indicator for the stream. For binary streams (all streams on ConvexOS are binary), it is the number of characters from the beginning of the stream.

rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, 0).

RETURN VALUE

If *fseek* is successful, a 0 is returned. *fseek* returns –1 for improper seeks. *fsetpos* returns 0 if successful and –1 on error. **errno** is set to **EBADF, ESPIPE,** or **EINVAL** on error.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- *ftell* does not always set **errno** on error.
- The *fgetpos* and *fsetpos* are not available in the backward-compatible mode.

SEE ALSO

lseek(2), fopen(3S)

NAME

ftime – get formatted date and time

SYNOPSIS

```
#include <sys/types.h>
#include <sys/timeb.h>
ftime(tp)
struct timeb *tp;
```

DESCRIPTION

This interface is obsoleted by *gettimeofday*(2).

The *ftime* function fills in a structure pointed to by its argument, as defined by *<sys/timeb.h>*:

```
/*      $CHeader: timeb.h 1.1 86/02/04 17:38:11 $*/
/*      Copyright 1984 Convex Computer Corp.*/

/*
 * Structure returned by ftime system call
 */
struct timeb
{
        time_t    time;
        unsigned short millitm;
        short     timezone;
        short     dstflag;
};
```

The structure contains the time since the epoch in seconds (up to 1000 milliseconds of more-precise interval), the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

SEE ALSO

date(1), gettimeofday(2), ctime(3), time(3)

NAME

gamma – log gamma function

SYNOPSIS

#include <math.h>

double gamma(x)
double x;

DESCRIPTION

Gamma returns ln $|\Gamma(|x|)|$. The sign of $\Gamma(|x|)$ is returned in the external integer *signgam*. The following C program might be used to calculate Γ:

```
y = gamma(x);
if (y > 88.0)
        error();
y = exp(y);
if(signgam < 0)
        y = -y;
```

DIAGNOSTICS

A huge value is returned for negative integer arguments.

BUGS

There should be a positive indication of error.

NAME

getactent, getactaid, getactnam, setactent, endactent – get activity file entry

SYNOPSIS

```
#include <act.h>

struct actstruct *getactent()

struct actstruct *getactaid(aid)
int aid;

struct actstruct *getactnam(name)
char *name;

setactent()

endactent()
```

DESCRIPTION

Getactent, getactaid, and *getactnam* each return pointers to an object with the following structure containing the broken-out fields of a line in the activities file.

```
/*      $CHeader: act.h 0.1 86/01/28 19:43:10 $*/
/*      Copyright 1984 Convex Computer Corp.*/
struct actstruct
{
    char    *act_name;      /* activity name */
    int     act_aid;        /* activity ID */
};

#define ACTFILE         "/etc/activities"
struct actstruct *getactent (), *getactnam (), *getactaid ();
```

Getactent simply reads the next line while *getactaid* and *getactnam* search until a matching *aid* or *name* is found (or until EOF is encountered). Each routine picks up where the others leave off so successive calls may be used to search the entire file. When using these routines it is not necessary to first open the file with *setactent* or to close the file with *endactent.*

A call to *setactent* has the effect of rewinding the activity file to allow repeated searches. *Endactent* may be called to close the activity file when processing is complete.

FILES

/etc/activities

SEE ALSO

activities(5),
"Accounting" chapter in the *CONVEX System Manager's Guide.*

DIAGNOSTICS

A null pointer (0) is returned on EOF or error.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getacwent, setacwent, endacwent – get actwho file entry

SYNOPSIS

#include <actwho.h>

struct actwho *getacwent()

setacwent()

endacwent()

DESCRIPTION

Getacwent returns a pointer to an object with the following structure containing the broken-out fields of a line in the group-activity access control file. This routine is useful for user programs which sort or summarize */etc/actwho*. An example of a program that would use *getacwent* is a program that reads the log file generated by the *bill* command every night and sorts the *actwho* file to optimize search time, putting entries referenced most often at the beginning of the *actwho* file.

```
/*      $CHeader: actwho.h 0.3 86/02/24 17:37:03 $*/
/*      Copyright 1984 Convex Computer Corp.*/
struct actwho {
        char        *acw_group;    /* project */
        char        *acw_activity; /* activity */
        char        **acw_members;/* list of users */
};

#define ACTWHOFILE              "/etc/actwho"
#define ACTWHOFILELOCK          "/etc/actwho.lock"
struct actwho                   *getacwent();
```

The members of this structure are:

acw_group
: The group name part of the group-activity combination for which access is being defined.

acw_activity
: The activity name part of the group-activity combination for which access is being defined.

acw_members
: Null-terminated vector of pointers to the individual member names.

Getacwent reads the next line in the *actwho* file until EOF is encountered. Each call picks up where the previous call left off, so successive calls may be used to search the entire file. When using this routine it is not necessary to open the file with *setacwent* or to close the file with *endacwent.*

A call to *setacwent* has the effect of rewinding the *actwho* file to allow repeated searches. *Endacwent* may be called to close the *actwho* file when processing is complete.

FILES

/etc/actwho

SEE ALSO

actwho(5),
"Accounting" chapter in the *CONVEX System Manager's Guide.*

DIAGNOSTICS

A null pointer (0) is returned on EOF or error.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved. Lines longer than 1024 characters can cause unpredictable results.

NAME

getc, getchar, fgetc, getw – get character or word from stream

SYNOPSIS

```
#include <stdio.h>

int getc(FILE *stream);

int getchar(void);

int fgetc(FILE *stream);

int getw(FILE *stream);
```

DESCRIPTION

getc returns the next character from the named input *stream*.

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

fgetc behaves like *getc*, but is a genuine function, not a macro; it may be used to save object text.

getw returns the next word from the named input *stream*. It returns the constant **EOF** upon end-of-file or error, but since that is a good integer value, *feof* and *ferror*(3S) should be used to check the success of *getw*. *getw* assumes no special alignment in the file.

SEE ALSO

fopen(3S), putc(3S), gets(3S), scanf(3S), fread(3S), ungetc(3S)

DIAGNOSTICS

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

A stop with message, 'Reading bad file', means an attempt has been made to read from a stream that has not been opened for reading by *fopen*.

BUGS

The end-of-file return from *getchar* is incompatible with that in UNIX editions 1-6.

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

To get *lint* to be quiet about references to *getc* and *getchar*, one must actually use the value that these functions return; using **void** will not work. Below is an example of a reference to *getc* that will keep *lint* happy:

```
if (getc (stdin) == EOF) {
        fprintf (stderr, "Could not getc from stdin.\n");
        exit (1);
}
```

NAME

getcwd – get the current working directory for the process

SYNOPSIS

```
char *getcwd(buf, size)
char *buf;
int size;
```

DESCRIPTION

The *getcwd()* function copies an absolute pathname of the current working directory into the character array pointed to by the argument *buf* and returns a pointer to the result. The *size* argument is the size in bytes of the character array pointed to by the *buf* argument. If *buf* is a **NULL** pointer, the behavior of *getcwd()* is undefined.

If successful, the *buf* pointer is returned. A **NULL** pointer is returned if an error occurs and the variable *errno* is set to indicate the error. The contents of *buf* after an error are undefined.

DIAGNOSTICS

If any of the following conditions occur, the *getcwd()* function returns a value of **NULL** and set *errno* to indicate the error.

[EINVAL] The *size* argument is less than or equal to zero.

[ERANGE] The *size* argument is greater than zero, but is smaller than the length of the pathname plus one.

[EACCES] Read or search permission was denied for a component of the pathname.

SEE ALSO

chdir(3)

NAME

getdiskbyname – get disk description by its name

SYNOPSIS

```
#include <disktab.h>

struct disktab *
getdiskbyname(name)
char *name;
```

DESCRIPTION

getdiskbyname takes a disk name (e.g. dkd-001) and returns a structure describing its geometry information and the standard disk partition tables. All information obtained from the *disktab*(5) file.

<disktab.h> has the following form:

```
/*      $CHeader: disktab.h 0.1 88/01/11 16:09:52 $     */
/*      Copyright 1984 Convex Computer Corp.*/

/*
 * Disk description table, see disktab(5)
 */
#define DISKTAB                 "/etc/disktab"

struct  disktab {
        char    *d_name;                /* drive name */
        char    *d_type;                /* drive type */
        int     d_secsize;              /* sector size in bytes */
        int     d_ntracks;              /* # tracks/cylinder */
        int     d_nsectors;             /* # sectors/track */
        int     d_ncylinders;           /* # cylinders */
        int     d_rpm;                  /* revolutions/minute */
        struct  partition {
                int     p_size;         /* #sectors in partition */
                int     p_bsize; /* block size in bytes */
                int     p_fsize; /* frag size in bytes */
        } d_partitions[8];
};

struct  disktab *getdiskbyname();
```

SEE ALSO

disktab(5)

DIAGNOSTICS

Returns NULL (0) if *name* is not found or an error occurs.

BUGS

This information should be obtained from the system for locally available disks (in particular, the disk partition tables). *getdiskbyname* returns a pointer to a static structure so it must be copied if it is to be saved.

NAME

getenv, setenv, unsetenv – manipulate environmental variables

SYNOPSIS

```
#include <stdlib.h>

char *getenv(const char *name);

int setenv(char *name, char *value, int overwrite);

void unsetenv(char *name);
```

DESCRIPTION

getenv searches the environment list (see *environ*(7)) for a string of the form *name=value* and returns a pointer to the string *value* if such a string is present, and 0 (NULL) if it is not.

setenv searches the environment list as *getenv* does; if the string *name* is not found, a string of the form *name=value* is added to the environment. If it is found, and *overwrite* is non-zero, its value is changed to *value*. *Setenv* returns 0 on success and -1 on failure, where failure is caused by an inability to allocate space for the environment.

unsetenv removes all occurrences of the string *name* from the environment. There is no library provision for completely removing the current environment. It is suggested that the following code be used to do so.

```
static char     *envinit[1];
extern char     **environ;
environ = envinit;
```

All of these routines permit, but do not require, a trailing equals ("=") sign on *name* or a leading equals sign on *value*.

COMPATIBILITY

When compiling or linking with the *-str* option of the **cc** command, different library routines are linked. In this mode, strict conformance to the ANSI C standard is specified. ANSI C reserves variable name *environ* for the C programmer. Another variable acceptable for use by the implementation is used instead.

SEE ALSO

environ(7), csh(1), sh(1), execve(2)

NAME

getfpmode – get current floating point mode

SYNOPSIS

#include <convex/fpmode.h>

int getfpmode(level)
int level;

DESCRIPTION

getfpmode returns the floating point mode that is in operation at the specified *level*. The integer value returned will be either FPMODE_IEEE or FPMODE_NATIVE, which are both defined in <*convex/fpmode.h*> and indicate the obvious setting of the IEEE bit in the PSW (on for IEEE, off for native).

If the argument, *level*, is 1, the mode for the current (calling) routine will be returned. For a *level* greater than 1, the *level*th ancestor routine will be checked and its mode returned.

SEE ALSO

setfpmode(3)

DIAGNOSTICS

A value of -1 is returned if an error is encountered.

NAME

getfsent, getfsspec, getfsfile, getfstype, setfsent, endfsent – get file system descriptor file entry

SYNOPSIS

```
#include <fstab.h>

struct fstab *getfsent()

struct fstab *getfsspec(spec)
char *spec;

struct fstab *getfsfile(file)
char *file;

struct fstab *getfstype(type)
char *type;

int setfsent()

int endfsent()
```

DESCRIPTION

These routines are included for compatibility with 4.2 BSD; they have been superseded by the *getmntent*(3) library routines.

getfsent, *getfsspec*, *getfstype*, and *getfsfile* each return a pointer to an object with the following structure containing the broken-out fields of a line in the file system description file, *<fstab.h>*.

```
struct fstab{
        char    *fs_spec;
        char    *fs_file;
        char    *fs_type;
        int     fs_freq;
        int     fs_passno;
};
```

The fields have meanings described in *fstab*(5).

getfsent reads the next line of the file, opening the file if necessary.

setfsent opens and rewinds the file.

endfsent closes the file.

getfsspec and *getfsfile* sequentially search from the beginning of the file until a matching special file name or file system file name is found, or until EOF is encountered. *getfstype* does likewise, matching on the file system type field.

NOTES

These routines only return file systems of type **4.2**, therefore, **nfs** entries are ignored.

FILES

/etc/fstab

SEE ALSO

fstab(5), getmntent(3)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

getgrent, setgrent, endgrent, fgetgrent – get group file entry

SYNOPSIS

#include <grp.h>

struct group *getgrent()

void setgrent()

void endgrent()

struct group *fgetgrent(f)
FILE *f;

DESCRIPTION

getgrent returns a pointer to an object with the following structure containing the broken-out fields of a line in the group file. Each line contains a "group" structure, defined in the *<grp.h>* header file.

```
struct group {
        char    *gr_name;
        char    *gr_passwd;
        int     gr_gid;
        char    **gr_mem;
};
```

The members of this structure are:

gr_name	The name of the group.
gr_passwd	The encrypted password of the group.
gr_gid	The numerical group ID.
gr_mem	A null-terminated array of pointers to the individual member names.

getgrent when first called returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls may be used to search the entire file. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. *endgrent* may be called to close the group file when processing is complete. Both *setgrent* and *endgrent* are void functions and therefore do not return a value.

fgetgrent returns a pointer to the next group structure in the stream *f*, which must refer to an open file in the same format as the group file */etc/group*. Yellow pages + or - entries are not expanded when using *fgetgrent*.

FILES

/etc/group
/etc/yp/*domainname*/group.byname
/etc/yp/*domainname*/group.bygid

SEE ALSO

getlogin(3), getgrgid(3), getpwent(3), group(5), ypserv(8), ypclnt(3N)

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

A valid line in the */etc/group* file must have a non-NULL group ID field. Any line that does not will be ignored for security reasons.

The above routines use *<stdio.h>* and Yellow Pages, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getgrgid, getgrnam – group database access routines

SYNOPSIS

#include <grp.h>

struct group *getgrgid(gid)
int gid;

struct group *getgrnam(name)
char *name;

DESCRIPTION

The *getgrgid()* and *getgrnam()* routines both return pointers to an object of type *struct group* containing an entry from the group database with the matching *gid* or *name*. This structure, which is defined in **<grp.h>**, includes the members shown below:

Member Type	Member Name	Description
char *	gr_name	the name of the group.
gid_t	gr_gid	the numerical group id.
char **	gr_mem	a NULL terminated vector of pointers to the individual member names.

A **NULL** pointer is returned on error or if the requested entry is not found. The return values may point to static data that is overwritten by each call.

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

The above routines use *<stdio.h>* and Yellow Pages, which causes them to increase the size of programs not using standard I/O more than might be expected.

BACKWARD COMPATIBILITY

See getgrent(3)

FILES

/etc/group
/etc/yp/*domainname*/group.byname
/etc/yp/*domainname*/group.bygid

SEE ALSO

getlogin(3), getpwuid(3), getgrent(3), getpwent(3)

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getlogin – get login name

SYNOPSIS

char *getlogin()

DESCRIPTION

getlogin returns a pointer to the login name as found in */etc/utmp*. It may be used in conjunction with *getpwuid* to locate the correct password file entry when the same userid is shared by several login names.

If *getlogin* is called within a process that is not attached to a terminal, it returns NULL. The correct procedure for determining the login name is to first call *getlogin* and if it fails, to call *getpwuid*(*getuid*()).

FILES

/etc/utmp

SEE ALSO

getpwent(3), getgrent(3), utmp(5)

DIAGNOSTICS

Returns NULL (0) if name not found.

BUGS

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

NAME

setmntent, getmntent, addmntent, endmntent, hasmntopt – get file system descriptor file entry

SYNOPSIS

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

FILE *setmntent(filep, type)
char *filep;
char *type;

struct mntent *getmntent(filep)
FILE *filep;

int addmntent(filep, mnt)
FILE *filep;
struct mntent *mnt;

char *hasmntopt(mnt, opt)
struct mntent *mnt;
char *opt;

int endmntent(filep)
FILE *filep;
```

DESCRIPTION

These routines replace the *getfsent* routines for accessing the file system description file */etc/fstab*. They are also used to access the mounted file system description file */etc/mtab*.

setmntent opens a file system description file and returns a file pointer which can then be used with *getmntent*, *addmntent*, or *endmntent*. The *type* argument is the same as in *fopen*(3). *getmntent* reads the next line from *filep* and returns a pointer to an object with the following structure containing the broken-out fields of a line in the filesystem description file, *<mntent.h>*. The fields have meanings described in *fstab*(5).

```
struct mntent {
        char    *mnt_fsname;    /* file system name */
        char    *mnt_dir;       /* file system path prefix */
        char    *mnt_type;      /* 4.2, nfs, swap, or xx */
        char    *mnt_opts;      /* ro, quota, etc. */
        int     mnt_freq;       /* dump frequency, in days */
        int     mnt_passno;     /* pass number on parallel fsck */
};
```

addmntent adds the *mntent* structure *mnt* to the end of the open file *filep*. Note that *filep* has to be opened for writing if this is to work. *hasmntopt* scans the *mnt_opts* field of the *mntent* structure *mnt* for a substring that matches *opt*. It returns the address of the substring if a match is found, 0 otherwise. *endmntent* closes the file.

FILES

/etc/fstab
/etc/mtab

SEE ALSO

fstab(5), mtab(5), getfsent(3X)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

The returned *mntent* structure points to static information that is overwritten in each call.

NAME

getopt – get option letter from argv

SYNOPSIS

int getopt(argc, argv, optstring)
int argc;
char **argv;
char *optstring;

extern char *optarg;
extern int optind;

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 *stderr* and returns a question mark (**?**) when it encounters an option letter not included in *optstring*.

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 int optind;
        extern char *optarg;
        .
        .
        .
        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 '?':
                default:
                        errflg++;
                        break;
                }
        if (errflg) {
                fprintf(stderr, "Usage: ...");
                exit(2);
        }
        for (; optind < argc; optind++) {
                .
                .
                .
        }
        .
        .
        .
}
```

HISTORY

Written by Henry Spencer, working from a Bell Labs manual page. Modified by Keith Bostic to behave more like the System V version.

BUGS

It is not obvious how '–' standing alone should be treated; this version treats it as a non-option argument, which is not always right.

Option arguments are allowed to begin with '–'; this is reasonable but reduces the amount of error checking possible.

getopt is quite flexible but the obvious price must be paid: there is much it could do that it doesn't, like checking mutually exclusive options, checking type of option arguments, etc.

NAME

getpass – read a password

SYNOPSIS

char *getpass(prompt)
char *prompt;

DESCRIPTION

Getpass reads a password from the file */dev/tty*, or if that cannot be opened, from the standard input, after prompting with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters.

FILES

/dev/tty

SEE ALSO

crypt(3)

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getpty – get a pseudo-teletype device name

SYNOPSIS

char *getpty()

DESCRIPTION

Getpty returns a pointer to the name of a pseudo-teletype device. It is preferable to use *getpty*, as it functions regardless of the number of pseudo-teletypes configured into the system, and independently of the arbitrary pseudo-teletype naming convention. The name *getpty* returns is an advisory name only, and does not guarantee that a call to *open*(2) with this name will succeed. A typical code sequence to obtain the master and slave file descriptors for a pseudo-teletype is as follows:

```
/*
 * open master side.
 */
for (;;) {
        if ((pty_name = getpty( )) == NULL)
            fatal ("Can't get a pty");

        if ((master_fd = open (pty_name, 2)) >= 0) {
            /*
             * open slave side.
             */
            pty_name[5] = 't';    /* /dev/ptyXY ===> /dev/ttyXY */
            if ((slave_fd = open (pty_name, 2)) >= 0)
                    break;
            else
                    close (master_fd);
        }
}
```

RETURN VALUE

If the call succeeds, a character pointer is returned which references the name of a valid pseudo-teletype device. If the call fails, NULL is returned, which indicates that there are no more pseudo-teletype devices.

SEE ALSO

open(2)

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getpw – get name from uid

SYNOPSIS

getpw(uid, buf)
char *buf;

DESCRIPTION

Getpw is obsoleted by getpwuid(3).

Getpw searches the password file for the (numerical) *uid*, and fills in *buf* with the corresponding line; it returns non-zero if *uid* could not be found. The line is null-terminated.

FILES

/etc/passwd

SEE ALSO

getpwent(3), passwd(5)

DIAGNOSTICS

Non-zero return on error.

NAME

getpwent, setpwent, endpwent, fgetpwent – get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent()

int setpwent()

int endpwent()

struct passwd *fgetpwent(f)
FILE *f;
```

DESCRIPTION

getpwent() returns a pointer to an object with the following structure containing the broken-out fields of a line in the password file. Each line in the file contains a "passwd" structure, declared in the *<pwd.h>* header file:

```
#define MAX_PWLEN  1024          /* max length of a passwd file entry */

struct  passwd {                 /* see getpwent(3) */
        char    *pw_name;
        char    *pw_passwd;
        int     pw_uid;
        int     pw_gid;
        int     pw_quota;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};
```

This structure is declared in *<pwd.h>* so it is not necessary to redeclare it.

The fields *pw_quota* and *pw_comment* are unused; the others have meanings described in *passwd*(5). When first called, *getpwent* returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; so successive calls can be used to search the entire file. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *endpwent* may be called to close the password file when processing is complete.

fgetpwent returns a pointer to the next passwd structure in the stream *f*, which matches the format of the password file */etc/passwd*. Yellow pages + or - entries are not expanded when using *fgetpwent*.

FILES

/etc/passwd
/etc/passwd.dir
/etc/passwd.pag
/etc/yp/*domainname*/passwd.byname
/etc/yp/*domainname*/passwd.byuid

SEE ALSO

getpwuid(3), getlogin(3), getgrent(3), passwd(5), ypserv(8), ypclnt(3N)

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

A valid line in the */etc/passwd* file must have non-NULL user ID and group ID fields. Any line that does not will be ignored for security reasons.

The above routines use *<stdio.h>* and Yellow Pages, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

The return value points to static information that is overwritten on each call.

NAME

getpwrestent, getpwrestuid, getpwrestnam, setpwrestent, endpwrestent, fgetpwrestent – get entry from password restrictions file

SYNOPSIS

```
#include <pwrest.h>

struct pwrestrict *getpwrestent()

struct pwrestrict *getpwrestuid(uid)
int uid;

struct pwrestrict *getpwrestnam(name)
char *name;

int setpwrestent()

int endpwrestent()

struct pwrestrict *fgetpwent(f)
FILE *f;
```

DESCRIPTION

getpwrestent, *getpwrestuid*, and *getpwrestnam* each return a pointer to an object with the following structure containing the broken-out fields of a line in the password restrictions file. Each line in the file contains a "pwrestrict" structure, declared in the *<pwrest.h>* header file:

```
/*      $CHeader: pwrest.h 0.1 87/01/29 13:48:57 $      */
/*      Copyright 1986 Convex Computer Corp.*/

struct          pwrestrict {      /* see getpwrestent(3) */
        char    *pwr_name;
        char    *pwr_passwd;
        char    *pwr_age;
        char    pwr_restrict;
        char    *pwr_usrrest;   /* unused */
        int     pwr_uid;
};

struct pwrestrict *getpwrestent(), *getpwrestuid(),
                  *getpwrestnam(), *fgetpwrestent();
```

This structure is declared in *<pwrest.h>* so it is not necessary to redeclare it.

The fields shown above have meanings described in *pwrestrict(5)*. When first called, *getpwrestent* returns a pointer to the first pwrestrict structure in the file; thereafter, it returns a pointer to the next pwrestrict structure in the file; so successive calls can be used to search the entire file. *getpwrestuid* searches the dbm(3) style password database if /etc/pwrestrict.dir and /etc/pwrestrict.pag are accessible, otherwise it searches from the beginning of the file until a numerical user id matching *uid* is found and returns a pointer to the particular structure in which it was found. You must pass the -ldbm option to the loader to use *getpwrestuid*. *getpwrestnam* searches the dbm(3) style password database if /etc/pwrestrict.dir and /etc/pwrestrict.pag are accessible, otherwise it searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found. You must pass the -ldbm option to the loader to use *getpwrestnam*. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setpwrestent* has the effect of rewinding the password restriction file to allow repeated searches. *endpwrestent* may be called to close the password file when processing is complete.

fgetpwrestent returns a pointer to the next pwrestrict structure in the stream *f*, which matches the format of the password file */etc/pwrestrict*. Yellow pages + or - entries are not expanded when using *fgetpwrestent*.

FILES

/etc/pwrestrict
/etc/pwrestrict.dir
/etc/pwrestrict.pag
/etc/yp/*domainname*/pwrestrict.byname
/etc/yp/*domainname*/pwrestrict.byuid

SEE ALSO

getlogin(3), getgrent(3), getpwent(3), passwd(5), pwrestrict(5), ypserv(8), ypclnt(3N)

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

A valid line in the */etc/pwrestrict* file must have a non-NULL user ID field. Any line that does not will be ignored for security reasons.

The above routines use *<stdio.h>* and Yellow Pages, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getpwuid, getpwnam – user database access

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwuid(uid)
int uid;

struct passwd *getpwnam(name)
char *name;
```

DESCRIPTION

The *getpwuid()* and *getpwnam()* functions both return a pointer to an object of type *struct passwd* containing an entry from the user database with a matching *uid* or *name*. This structure, which is defined in **<pwd.h>**, includes the members shown below:

Member Type	Member Name	Description
char *	pw_name	the users login name.
uid_t	pw_uid	the users id number.
gid_t	pw_gid	the users group id number.
char *	pw_dir	the users initial working directory (home directory.)
char *	pw_shell	the users initial program (shell.)

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNINGS

The above routines use **<stdio.h>** and Yellow Pages, which causes them to increase the size of programs not using standard I/O more than might be expected.

The implementation of the *cuserid()* function uses the *getpwnam()* function; thus the results of a user's call to either routine will be overwritten by a subsequent call to the other routine.

BACKWARDS COMPATIBILITY

See getpwent(3)

FILES

/etc/passwd
/etc/passwd.dir
/etc/passwd.pag
/etc/yp/*domainname*/passwd.byname
/etc/yp/*domainname*/passwd.byuid

SEE ALSO

getlogin(3), cuserid(3), getgrgid(3), getpwent(3), getgrent(3)

BUGS

The return value points to static information that is overwritten on each call.

NAME

gets, fgets – get a string from a stream

SYNOPSIS

#include <stdio.h>

char *gets(char *s);

char *fgets(char *s, int n, FILE *stream);

DESCRIPTION

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

fgets reads *n*–1 characters, or up to a newline character, whichever comes first, from the *stream* into the string *s*. The last character read into *s* is followed by a null character. *fgets* returns its first argument.

SEE ALSO

puts(3S), getc(3S), scanf(3S), fread(3S), ferror(3S)

DIAGNOSTICS

gets and *fgets* return the constant pointer NULL upon end of file or error.

BUGS

gets deletes a newline, *fgets* keeps it, all in the name of backward compatibility.

NAME

getshares - get shares file entry given uid

SYNOPSIS

```
#include <shares.h>

unsigned long getshares(lp, uid, lock)
struct lnode * lp;
int uid;
int lock;
```

DESCRIPTION

getshares finds the shares entry with the same uid as *uid* and reads it into an area pointed to by *lp*. *lock* should be non-zero if the shares data-base should be opened for writing.

getshares returns zero if no entry exists, (in which case all non-*uid* fields of **lp* will have been cleared), or if uid is greater than MAXUID. *getshares* will return 1 if the "last used" time of the entry is equal to zero; otherwise, *getshares* returns the "last used" time of the entry.

lp->l_uid will always have been set to *uid* on return.

FILES

/etc/shares	share data-base file
/usr/lib/libshare.a	the share-specific library routines

SEE ALSO

closeshares(3), getshput(3), openshares(3), putshares(3), sharesfile(3)

DIAGNOSTICS

getshares returns 0 if an entry cannot be found. *getshares* returns -1 if a system error occurred and the external variable *errno* is set to indicate the cause of the error.

NOTE

The Share scheduler is an optional product; for more information, contact your CONVEX sales representative.

NAME

getshput - read, modify, and write shares file entry

SYNOPSIS

```
#include <shares.h>

unsigned long getshput(lp, func)
struct lnode * lp;
unsigned long (*func)();
```

DESCRIPTION

getshput finds the shares entry with the same uid as *lp->l_uid* and reads it into an internal *lnode* structure. The function pointed to by *func* is then called with two arguments, the first pointing to the original *lnode* structure, and the second pointing to the internal *lnode* structure.

If *func* returns with a value other than 0 the (presumably modified) second structure is written back to the shares file, with the "extime" field in the share record filled with the value returned by *func*. If the write fails, then *getshput* will return -1.

If *func* returns zero, *getshput* simply returns.

getshput returns the value returned by *func*, or -1 if the write to the shares file fails.

FILES

/etc/shares	share data-base file
/usr/lib/libshare.a	the share-specific library routines

SEE ALSO

closeshares(3), getshares(3), openshares(3), putshares(3), sharesfile(3)

DIAGNOSTICS

getshput returns 0 if the entry can't be modified.

NOTE

The Share scheduler is an optional product; for more information, contact your CONVEX sales representative.

NAME

getttyent, getttynam, setttyent, endttyent – get ttys file entry

SYNOPSIS

#include <ttyent.h>

struct ttyent *getttyent()

struct ttyent *getttynam(name)
char *name;

setttyent()

endttyent()

DESCRIPTION

Getttyent, and *getttynam* each return a pointer to an object with the following structure containing the broken-out fields of a line from the tty description file.

```
/*      $CHeader: ttyent.h 0.1 89/10/04 16:10:51 $      */
/*      Copyright 1989 Convex Computer Corp.*/
/* $Cheader : $
 */
struct  ttyent { /* see getttyent(3) */
        char    *ty_name;       /* terminal device name */
        char    *ty_getty;      /* command to execute, usually getty */
        char    *ty_type;       /* terminal type for termcap (3X) */
        int     ty_status;      /* status flags (see below for defines) */
        char    *ty_window;     /* command to start up window manager */
        char    *ty_comment;.   /* usually the location of the terminal */
};

#define TTY_ON                  0x1     /* enable logins (startup getty) */
#define TTY_SECURE 0x2          /* allow root to login */
#define TTY_DIALUP  0x4         /* check dialup password */
#define TTY_RESTRICTED  0x8     /* check if there are any access or deny */
#define TTY_UUCP    0x10        /* check uucp only */
extern struct ttyent *getttyent();
extern struct ttyent *getttynam();
```

ty_name is the name of the character-special file in the directory "/dev". For various reasons, it must reside in the directory "/dev".

ty_getty is the command (usually *getty*(8)) which is invoked by *init* to initialize tty line characteristics. In fact, any arbitrary command can be used; a typical use is to initiate a terminal emulator in a window system.

ty_type is the name of the default terminal type connected to this tty line. This is typically a name from the *termcap*(5) data base. The environment variable 'TERM' is initialized with this name by *getty*(8) or *login*(1).

ty_status is a mask of bit fields which indicate various actions to be allowed on this tty line. The following is a description of each flag.

TTY_ON — Enables logins (i.e., *init*(8) will start the specified "getty" command on this entry).

TTY_SECURE — Allows root to login on this terminal. Note that 'TTY_ON' must be included for this to be useful.

TTY_DIALUP — Indicates that this a dialup terminal and a dialup password is required for login. Note that 'TTY_ON' must be

included for this to be useful.

TTY_RESTRICTED — Indicates that this terminal is restricted for login by some users. The restriction list is specified in the */etc/ttys* file. Note that 'TTY_ON' must be included for this to be useful.

ty_window — is the command to execute for a window system associated with the line. The window system will be started before the command specified in the *ty_getty* entry is executed. If none is specified, this will be null.

ty_comment — is the trailing comment field, if any; a leading delimiter and white space will be removed.

Getttyent reads the next line from the ttys file, opening the file if necessary; *setttyent* rewinds the file; *endttyent* closes it.

Getttynam searches from the beginning of the file until a matching *name* is found (or until EOF is encountered).

FILES

/etc/ttys

SEE ALSO

login(1), ttyslot(3), ttys(5), gettytab(5), termcap(5), getty(8), init(8)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

getusershell, setusershell, endusershell – get legal user shells

SYNOPSIS

char *getusershell()

setusershell()

endusershell()

DESCRIPTION

Getusershell returns a pointer to a legal user shell as defined by the system manager in the file */etc/shells*. Successive calls to *getusershell* return successive entries from /etc/shells. NULL is returned on calls made after the last entry has been returned. If */etc/shells* does not exist, *getusershell* returns in succession the two standard system shells */bin/sh* and */bin/csh*.

Getusershell reads the next line (opening the file if necessary); *setusershell* rewinds the file; *endusershell* closes it.

FILES

/etc/shells

SEE ALSO

shells(5)

DIAGNOSTICS

The routine *getusershell* returns a null pointer (0) on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

getwd – get current working directory pathname

SYNOPSIS

char *getwd(pathname)
char *pathname;

DESCRIPTION

Getwd copies the absolute pathname of the current working directory to *pathname* and returns a pointer to the result.

LIMITATIONS

Maximum pathname length is MAXPATHLEN characters (1024).

DIAGNOSTICS

Getwd returns zero and places a message in *pathname* if an error occurs.

BUGS

Getwd may fail to return to the current directory if an error occurs.

NAME

hypot, cabs, shypot, scabs – Euclidean distance

SYNOPSIS

```
#include <math.h>

double hypot(x, y)
double x, y;

double cabs(z)
struct { double x, y;} z;

float shypot(x, y)
float x, y;

float scabs(z)
struct { float x, y;} z;
```

DESCRIPTION

Hypot, shypot, cabs and *scabs* return

sqrt(x*x + y*y),

taking precautions against unwarranted overflows.

SEE ALSO

exp(3M) for *sqrt*

NAME

initgroups – initialize group access list

SYNOPSIS

initgroups(name, basegid)
char *name;
int basegid;

DESCRIPTION

Initgroups reads through the group file and sets up, using the *setgroups*(2) call, the group access list for the user specified in *name*. The *basegid* is automatically included in the groups list. Typically this value is given as the group number from the password file.

FILES

/etc/group

SEE ALSO

setgroups(2)

DIAGNOSTICS

Initgroups returns –1 if it was not invoked by the super-user.

BUGS

Initgroups uses the routines based on *getgrent*(3). If the invoking program uses any of these routines, the group structure will be overwritten in the call to *initgroups*.

No one seems to keep /etc/group up to date.

NAME

insque, remque – insert/remove element from a queue

SYNOPSIS

```
struct qelem {
        struct  qelem *q_forw;
        struct  qelem *q_back;
        char    q_data[];
};

insque(elem, pred)
struct qelem *elem, *pred;

remque(elem)
struct qelem *elem;
```

DESCRIPTION

Insque and *remque* manipulate queues built from doubly linked lists. Each element in the queue must in the form of "struct qelem". *Insque* inserts *elem* in a queue immediately after *pred*; *remque* removes an entry *elem* from a queue.

NAME

j0, j1, jn, y0, y1, yn – bessel functions

SYNOPSIS

```
#include <math.h>

double j0(x)
double x;

double j1(x)
double x;

double jn(n, x)
double x;

double y0(x)
double x;

double y1(x)
double x;

double yn(n, x)
double x;
```

DESCRIPTION

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

DIAGNOSTICS

Negative numbers cause *y0, y1,* and *yn* to return a huge negative value and set *errno* to EDOM.

NAME

limits.h – header file contain numerical limits

SYNOPSIS

#include <limits.h>

DESCRIPTION

limits.h contains information on the numerical limits of the system.

CHAR_BIT is the number of bits in a byte.

SCHAR_MIN, SCHAR_MAX, UCHAR_MIN, UCHAR_MAX, CHAR_MIN and **CHAR_MAX** are the minimum and maximun value of signed characters, unsigned characters and plain characters, respectively.

SHRT_MIN, SHRT_MAX, INT_MIN, INT_MAX, LONG_MIN, LONG_MAX, LONG_LONG_MIN, and **LONG_LONG_MAX** are the minimum and maximum values that can be acquired by objects of type **short int, int, long int,** and **long long int** , respectively. **USHRT_MAX, UINT_MAX, ULONG_MAX** and **ULONG_LONG_MAX** are the maximum values for the unsigned integral types.

NAME

lockf – advisory record locking on files

SYNOPSIS

```
#include <fcntl.h>

#define     F_ULOCK     0     /* Unlock a previously locked section */
#define     F_LOCK      1     /* Lock a section for exclusive use */
#define     F_TLOCK     2     /* Test and lock a section (non-blocking) */
#define     F_TEST      3     /* Test section for other process' locks */

lockf(fd, cmd, size)
int fd, cmd;
long size;
```

DESCRIPTION

lockf may be used to test, apply, or remove an *advisory* record lock on the file associated with the open descriptor *fd*. (See *fcntl*(2) for more information about advisory record locking.)

A lock is obtained by specifying a *cmd* parameter of F_LOCK or F_TLOCK. To unlock an existing lock, the F_ULOCK *cmd* is used. F_TEST is used to detect if a lock by another process is present on the specified segment.

F_LOCK and F_TLOCK requests differ only by the action taken if the lock may not be immediately granted. F_TLOCK will cause the function to return a –1 and set *errno* to EAGAIN if the section is already locked by another process. F_LOCK will cause the process to sleep until the lock may be granted or a signal is caught.

size is the number of contiguous bytes to be locked or unlocked. The lock starts at the current file offset in the file and extends forward for a positive *size* or backward for a negative *size* (preceding but not including the current offset). A segment need not be allocated to the file in order to be locked; however, a segment may not extend to a negative offset relative to the beginning of the file. If *size* is zero, the lock will extend from the current offset through the end-of-file. If such a lock starts at offset 0, then the entire file will be locked (regardless of future file extensions).

NOTES

The descriptor *fd* must have been opened with O_WRONLY or O_RDWR permission in order to establish locks with this function call.

All locks associated with a file for a given process are removed when the file is closed or the process terminates. Locks are not inherited by the child process in a *fork*(2) system call.

RETURN VALUE

Zero is returned on success, –1 on error, with an error code stored in *errno*.

ERRORS

lockf will fail if one or more of the following are true:

[EBADF]	*fd* is not a valid open descriptor.
[EBADF]	*cmd* is F_LOCK or F_TLOCK and the process does not have write permission on the file.
[EAGAIN]	*cmd* is F_TLOCK or F_TEST and the section is already locked by another process.
[EINTR]	*cmd* is F_LOCK and a signal interrupted the process while it was waiting for the lock to be granted.
[EDEADLK]	*cmd* is F_LOCK, F_TLOCK, or F_ULOCK and there are no more file lock entries available.
[EINVAL]	The specified *cmd* is not one of the supported ones listed above.

[EACCES] The *cmd* is F_TEST and the specified region (or part of it) is exclusively locked by another process.

SEE ALSO

fcntl(2), lockd(8C)

BUGS

File locks obtained through the *lockf* mechanism do not interact in any way with those acquired via *flock*(2). They do, however, work correctly with the locks claimed by *fcntl*(2).

NAME

malloc, free, realloc, calloc, cfree, alloca – memory allocator

SYNOPSIS

#include <stdlib.h>

void *malloc(size_t size);

void free(void *ptr);

void *realloc(void *ptr, size_t size);

void *calloc(size_t nelem, size_t elsize);

int cfree(int *ptr, unsigned int nelem, unsigned int elsize);

char *alloca(int size);

DESCRIPTION

malloc and *free* provide a general-purpose memory allocation package. *malloc* 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*.

malloc maintains multiple lists of free blocks according to size, allocating space from the appropriate list. It calls *sbrk* (see *brk*(2)) to get more memory from the system when there is no suitable space already free.

realloc 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.

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.

calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros. *cfree* frees space previously allocated by *calloc*.

alloca allocates *size* bytes of space in the stack frame of the caller. This temporary space is automatically freed on return.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object. If the space is of *pagesize* or larger, the memory returned will be page-aligned.

SEE ALSO

brk(2), getpagesize(2)

DIAGNOSTICS

malloc, *realloc* and *calloc* return a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. *malloc* may be recompiled to check the arena very stringently on every transaction; those sites with a source code license may check the source code to see how this can be done.

BUGS

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

The current implementation of *malloc* does not always fail gracefully when system memory limits are approached. It may fail to allocate memory when larger free blocks could be broken up, or when limits are exceeded because the size is rounded up. It is optimized for sizes that are powers of two.

alloca is machine dependent; its use is discouraged.

NAME

mblen, mbtowc, wctomb, mbstowcs, wcstombs – multibyte and wide character functions

SYNOPSIS

#include <stdlib.h>

int mblen(const char *s, size_t n);

int mbtowc(wchar_t *pwc, const char *s, size_t n);

int wctomb(char *s, wchar_t wchar);

size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);

size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);

DESCRIPTION

These functions are of limited usefulness because CONVEX C supports only the "C" locale. If **s* is **NULL** in any of the following functions, 0 is returned indicating that no state-dependent encodings exist. The conversion between multibyte character strings and wide characters are trivial because no special encodings are supplied by CONVEX.

mblen returns the number of bytes, up to *n*, of the multibyte character pointed to by *s*. This is always 1 unless $n = 0$ or **s* == **NULL**.

mbtowc converts a multibyte string to a wide character. Because all multibyte characters are length 1, the first character pointed to by **s* is copied to **pwc* and 1 is returned. However, if $n = 0$, no conversion is performed and -1 is returned.

wctomb converts a wide character to a multibyte character code stored in the object *wchar* into a multibyte character whose base address is *s*. 1 is returned because all multibyte characters are of length 1.

mbstowcs copies *n* bytes, or until **NULL** is encountered, of the multibyte string pointed to by **s* to the wide character string pointed to by **pwcs*. Returns the number of multibyte characters copied.

wcstombs copies an array of *n* or fewer multibyte character encodings into a multibyte character string. Returns one less than the number of characters copied.

NOTES

These functions are intended for use on systems implementing locales with extended character sets and are provided for standard conformance.

NAME

mkfifo – make a FIFO special file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(path, mode)
char *path;
mode_t mode;
```

DESCRIPTION

mkfifo() creates a new FIFO special file with name *path*. The mode of the new FIFO special file is initialized from *mode*.

The FIFO's owner ID is set to the process's effective user ID. The FIFO's group ID is set to that of the parent directory in which it is created.

The low-order 9 bits of mode are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared.

RETURN VALUE

A 0 return value indicates success. A –1 return value indicates an error, and an error code is stored in *errno*.

ERRORS

mkfifo() will fail and no FIFO special file will be created if:

[EACCES] Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the FIFO special file to be created.

[EEXIST] The named file exists.

[EINVAL] The *path* argument contains a byte with the high-order bit set.

[ENAMETOOLONG] The length of *path* exceeds PATH_MAX for the file system.

[ENAMETOOLONG] The length of a pathname component exceeds NAME_MAX and the file system enforces _POSIX_NO_TRUNC.

[ENOENT] A component of the path prefix does not exist.

[ENOENT] The *path* argument points to the empty string.

[ENOSPC] The file system does not contain enough space to expand the parent directory.

[ENOTDIR] A component of the path prefix is not a directory.

[EROFS] The named file resides on a read-only file system.

[EFAULT] *path* points outside the process's allocated address space.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EIO] An I/O error occurred while writing to the file system.

SEE ALSO

chmod(2), stat(2), umask(2), mknod(2)

NAME

mktemp, mkstemp – make a unique file name

SYNOPSIS

char *mktemp(template)
char *template;

mkstemp(template)
char *template;

DESCRIPTION

mktemp creates a unique file name, typically in a temporary filesystem, by replacing *template* with a unique file name, and returns the address of the template. The template should contain a file name with six trailing X's, which are replaced with the current process id and a unique letter. *mktemp* does not create a file, rather it checks for the existence of a file with the specified name. It is the user's responsibility to create a file with the name supplied by *mktemp*.

mkstemp makes the same replacement to the template but returns a file descriptor for the template file open for reading and writing. *mkstemp* avoids the race between testing whether the file exists and opening it for use.

SEE ALSO

getpid(2), open(2)

DIAGNOSTICS

mkstemp returns an open file descriptor upon success. It returns -1 if no suitable file could be created.

NAME

monitor, monstartup, moncontrol – prepare execution profile

SYNOPSIS

#include <mon.h>

monitor(*lowpc, highpc, buffer, bufsize, nfunc* **)**
int (**lowpc***)(), (****highpc***)();**
WORD buffer[];

monstartup(*lowpc, highpc***)**
int (**lowpc***)(), (****highpc***)();**

moncontrol(*mode***)**

DESCRIPTION

There are two different forms of monitoring available: An executable program created by:

cc –p . . .

automatically includes calls for the *prof*(1) monitor and includes an initial call to its start-up routine *monstartup* with default parameters; *monitor* need not be called explicitly except to gain fine control over profile buffer allocation. An executable program created by:

cc –pg . . .

automatically includes calls for the *gprof*(1) monitor.

Monstartup is a high-level interface to *lprofil*(2). *Lowpc* and *highpc* specify the address range that is to be sampled; the lowest address sampled is that of *lowpc* and the highest is just below *highpc*. *Monstartup* allocates space using *sbrk*(2) and passes it to *monitor* (see below) to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. Only calls of functions compiled with the profiling option **–p** of *cc*(1) are recorded.

To profile the entire program, it is sufficient to use

```
extern etext();
. . .
monstartup(0x80001000, etext);
```

Etext lies just above all the program text, see *end*(3).

To stop execution monitoring and write the results on the file *mon.out,* use

monitor(0);

then *prof*(1) can be used to examine the results.

Moncontrol is used to selectively control profiling within a program. This works with either *prof*(1) or *gprof*(1) type profiling. When the program starts, profiling begins. To stop the collection of histogram ticks and call counts, use *moncontrol*(0); to resume the collection of histogram ticks and call counts, use *moncontrol*(1). This allows the cost of particular operations to be measured. Note that an output file will be produced upon program exit regardless of the state of *moncontrol.*

Monitor is a low-level interface to *lprofil*(2). *Lowpc* and *highpc* are two address in the executable code and specify the range in which profiling si to be done; *buffer* is the address of a (user-supplied) memory buffer which must be large enough for a header of type *struct phdr* defined in *(mon.h)* , an array of *nfunc struct cnt* defined in *(mon.h)* in which the profiler will count calls to routines compiled with –p or –pg, and an array of *bufsize WORD* defined in *(mon.h).* At most *nfunc* call counts can be kept. For the results to be significant, especially where there are small,

heavily used routines, it is suggested that the *WORD* part of the buffer be no more than a few times smaller than the range of locations sampled. *Monitor* divides the buffer into space to record the histogram of program counter samples over the range *lowpc* to *highpc*, and space to record call counts of functions compiled with the **-p** option to *cc*(1).

To profile the entire program, it is sufficient to use

```
extern etext();
. . .
monitor(0x80001000, etext, buf, bufsize, nfunc);
```

FILES

mon.out

SEE ALSO

cc(1), prof(1)(optional product), gprof(1)(optional product), lprofil(2), brk(2)

NAME

mset, mclear – shared memory synchronization primitives

SYNOPSIS

```
#include      <sys/types.h>
mset(sem,wait)
semaphore *sem;
int wait;
mclear(sem)
semaphore *sem;
```

DESCRIPTION

These routines provide interprocess synchronization for shared memory.

mset indivisibly tests and sets a lock in semaphore *sem*. If the previous value of the lock was zero, the calling process has acquired the lock and *mset* returns true (**1**) immediately. If the previous value of the lock was non-zero, and the *wait* flag was set, *mset* will relinquish the processor until some other process releases the lock, at which time it again trys to acquire the lock. If the previous value of the lock was non-zero, and the *wait* flag is zero, *mset* returns failure (**0**).

mclear indivisibly clears the lock in *sem*, and awakens all other processes waiting on the semaphore.

sem is a pointer to a machine dependent semaphore structure, defined in *sys/types.h*. On the Convex Computer, the structure is:

```
struct      semaphore   {     /* machine dependent */
      char  lock;       /* the tas lock */
      char  awakened;   /* waiters have been awakened */
      };
#typedef struct semaphore semaphore;
```

sem must be in a shared memory region, with at least **PROT_READ** and **PROT_WRITE**. The **MAP_HASSEMAPHORE** flag must have been set when the *mmap* call was executed. The region may be mapped in any type– MAP_FILE, MAP_ANON, or MAP_DEVICE.

RETURN VALUE & ERRORS

mset returns **1** if the lock was acquired; **0** if not. A **-1** is returned if a system call internal to *mset* returned an error. *mclear* returns a positive value or zero if successful; a **-1** is returned if a system call internal to *mclear* returned an error. Note that *mset* is interruptible, and may return **-1** with *errno* set to EINTR if a signal is received during the sleep.

SEE ALSO

mmap(2), msleep(2), tas(3)

NAME

dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr – data base subroutines

SYNOPSIS

```
#include <ndbm.h>

typedef struct {
    char *dptr;
    int dsize;
} datum;

DBM *dbm_open(file, flags, mode)
    char *file;
    int flags, mode;

void dbm_close(db)
    DBM *db;

datum dbm_fetch(db, key)
    DBM *db;
    datum key;

int dbm_store(db, key, content, flags)
    DBM *db;
    datum key, content;
    int flags;

int dbm_delete(db, key)
    DBM *db;
    datum key;

datum dbm_firstkey(db)
    DBM *db;

datum dbm_nextkey(db)
    DBM *db;

int dbm_error(db)
    DBM *db;

int dbm_clearerr(db)
    DBM *db;
```

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. This package replaces the earlier *dbm*(3x) library, which managed only a single database.

Keys and *contents* are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has '.dir' as its suffix. The second file contains all data and has '.pag' as its suffix.

Before a database can be accessed, it must be opened by *dbm_open*. This will open and/or create the files *file*.**dir** and *file*.**pag** depending on the flags parameter (see *open*(2)).

Once open, the data stored under a key is accessed by *dbm_fetch* and data is placed under a key by *dbm_store*. The *flags* field can be either **DBM_INSERT** or **DBM_REPLACE**. **DBM_INSERT** will only insert new entries into the database and will not change an existing entry with the same key. **DBM_REPLACE** will replace an existing entry if it has the same key. A key (and its associated contents) is deleted by *dbm_delete*. A linear pass through all keys in a

database may be made, in an (apparently) random order, by use of *dbm_firstkey* and *dbm_nextkey*. *Dbm_firstkey* will return the first key in the database. *Dbm_nextkey* will return the next key in the database. This code will traverse the data base:

for (key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))

Dbm_error returns non-zero when an error has occurred reading or writing the database. *Dbm_clearerr* resets the error condition on the named database.

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates ok. Routines that return a *datum* indicate errors with a null (0) *dptr*. If *dbm_store* called with a *flags* value of **DBM_INSERT** finds an existing entry with the same key it returns 1.

BUGS

The '.pag' file will contain holes so that its apparent size is about four times its actual content. Older UNIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (cp, cat, tp, tar, ar) without filling in the holes.

Dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls. This storage is not necessarily aligned; stored "longs", for example, should be copied to a properly aligned block of memory before being accessed.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 4096 bytes). Moreover all key/content pairs that hash together must fit on a single block. *Dbm_store* will return an error in the event that a disk block fills with inseparable data.

Dbm_delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by *dbm_firstkey* and *dbm_nextkey* depends on a hashing function, not on anything interesting.

SEE ALSO

dbm(3X)

NAME

nfabort – dump core and log it in a notesfile

SYNOPSIS

nfabort (notesfile, message, title, cname, exitcode)
char *notesfile, *message, *title, *cname, exitcode

cc ... –lnfcom

DESCRIPTION

Nfabort provides user programs with a convenient way to generate a core image, move it to a save place, and log the action in a notesfile.

The *notesfile* parameter specifies the notesfile to receive a copy of the string *message* with a line appended detailing the final resting place of the core image. The text is inserted into the notesfile as a base note with *title* taken from the parameter list. *Cname* is the prefix of the pathname of where to place the core image. This is suffixed with ".integer" to yield the full pathname. The integer is generated from the pid of the current process.

After generating and saving the core image and placing the message in the notesfile, *nfabort* terminates the current process with the exit code specified by the *exitcode* parameter.

Nfabort calls *nfcomment* to insert the message into the notesfile.

BUGS

Certain conditions, such as running out of memory, will cause *nfabort* to fail.

Nfabort will fail to log the message if it can't fork a child process.

The final resting place of the core image will not be logged if *nfabort* can't allocate memory for temporary strings.

FILES

/usr/lib/libnfcom.a -lnfcom library

SEE ALSO

malloc(3), nfcomment(3), nfpipe(1), notes(1), popen(3S), system(3),
"The Notesfile Reference Manual" in the *CONVEX UNIX Tutorial Papers*

AUTHORS

Ray Essick

NAME

nfcomment – a user interface to the notesfile system

SYNOPSIS

nfcomment (notesfile, text, title, dirflag, anonflag)
char *nfname, *text, *title;

cc ... –lnfcom

DESCRIPTION

Nfcomment provides user programs with the ability to insert notes into a notesfile.

The note is inserted into the notesfile specified by *nfname*. *Text* is the address of the body of the note; this must be null-terminated. If *text* is NULL, the note is gathered from standard input until an EOF is encountered. The note is entered with the title specified by the *title* parameter. If the *dirflag* or *anonflag* parameters are non-zero, the director message is enabled or the note is entered anonymously. These take effect only if the user has the appropriate privileges in the notesfile.

Nfpipe is used to make the actual insertion of the text.

FILES

/usr/lib/libnfcom.a -lnfcom library

SEE ALSO

nfpipe(1), notes(1), popen(3S), system(3),
"The Notesfile Reference Manual" in the *CONVEX UNIX Tutorial Papers*

AUTHORS

Ray Essick

Rob Kolstad

NAME

nice – set program priority

SYNOPSIS

nice(incr)

DESCRIPTION

This interface is obsoleted by setpriority(2).

The scheduling priority of the process is augmented by *incr*. Positive priorities get less service than normal. Priority 10 is recommended to users who wish to execute long-running programs without flak from the administration.

Negative increments are ignored except on behalf of the superuser. The priority is limited to the range –64 (most urgent) to 64 (least).

The priority of a process is passed to a child process by *fork*(2). For a privileged process to return to normal priority from an unknown state, the system call *setpriority*(2) should be used; *nice* should **not** be used.

SEE ALSO

nice(1), setpriority(2), fork(2), renice(8)

NAME

nlist – get entries from name list

SYNOPSIS

#include <nlist.h>

nlist(filename, nl)
char *filename;
struct nlist nl[];

DESCRIPTION

Nlist examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See *a.out*(5) for the structure declaration.

This subroutine is useful for examining the system name list kept in the file **/vmunix**. In this way programs can obtain system addresses that are up to date.

SEE ALSO

a.out(5)

DIAGNOSTICS

All type entries are set to 0 if the file cannot be found or if it is not a valid namelist.

NAME

openshares – open share data base file

SYNOPSIS

int openshares(lock)
int lock;

DESCRIPTION

If *lock* is 1, *openshares* attempts to open the share data base file for writing, otherwise, if *lock* is 0, it attempts to open it for reading.

FILES

/etc/shares share data-base file

SEE ALSO

closeshares(3), getshares(3), getshput(3), putshares(3), sharesfile(3)

DIAGNOSTICS

If the opens succeeds, it returns 1. If the user does not have write access to the share data base file, *openshares* returns -1. If the user does not have read access to share data base file, it returns 0.

If the shares file is already open, this routine does nothing and returns a value of 1.

NOTE

The Share scheduler is an optional product; for more information, contact your CONVEX sales representative.

NAME

pathconf, fpathconf – configurable pathname variables

SYNOPSIS

```
#include <unistd.h>

long pathconf(path, name)
char *path;
int name;

long fpathconf(fildes, name)
int fildes, name;
```

DESCRIPTION

The *pathconf()* and *fpathconf()* functions obtain information about configurable limits or options associated with a file or directory.

The values that may be interrogated follow.

_PC_LINK_MAX
: Maximum number of hard links to a file (see *link(2)*).

_PC_MAX_CANON
: Maximum number of bytes in a terminal canonical queue.

_PC_MAX_INPUT
: Minimum number of bytes which will be available in a terminal input queue; hence, the maximum number of bytes an application may require to be typed as input before reading them.

_PC_NAME_MAX
: Maximum number of bytes in a file name, not including the terminating null.

_PC_PATH_MAX
: Maximum number of bytes in a pathname, excluding a terminating null.

_PC_PIPE_BUF Maximum number of bytes that may be atomically written to a pipe.

_PC_CHOWN_RESTRICTED
: If true, the use of *chown(2)* to change a file's ownership is restricted to root, and changing the group ownership is restricted to a group in the caller's supplementary group ID list.

_PC_NO_TRUNC
: If true, pathnames longer than _PC_NAME_MAX generate an error.

_PC_VDISBALE The value to use in disabling special character processing for a terminal; see *tcgetattr(3)* and *tcsetattr(3)*.

RETURN VALUE

Upon successful completion, the value of the requested *name* is returned without changing *errno*. The value returned will be no more restrictive than the corresponding compile-time value from **<limits.h>**.

If *name* is an invalid value, –1 is returned.

If *name* has no limit for the path or file descriptor, the return is –1 and *errno* is unchanged.

ERRORS

pathconf() and *fpathconf()* will fail if one or more of the following are true:

[EINVAL] The value of *name* is invalid.

pathconf() fails if:

[EACCES] Search permission is denied for a component of the path prefix of *path*.

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG]
The length of *path* exceeds PATH_MAX, or the length of a component is greater than NAME_MAX for a file system with NO_TRUNC in effect.

[ENOTDIR] A component of the path prefix of *path* is not a directory.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[EFAULT] *path* points to an invalid address.

[EIO] An I/O error occurred while reading from or writing to the file system.

fpathconf() will fail if one or more of the following are true:

[EBADF] *fildes* is not a valid open file descriptor.

SEE ALSO

fstab(5), mtab(5), mount(1)

NOTES

The application is at the mercy of the system administrator's thoroughness in setting up /etc/fstab to accurately describe any non-BSD-style file systems that are remote-mounted.

The values returned by *pathconf()* for _PC_MAX_INPUT and _PC_MAX_CANON are a minimal approximation; it is recommended that the user call the *fpathconf()* function after opening the device to obtain accurate measures of serial device buffer sizes.

NAME

pause – stop until signal

SYNOPSIS

int pause()

DESCRIPTION

pause() never returns normally. It is used to give up control while waiting for a signal from *kill*(2) or an interval timer, such as *alarm(3c)*. Upon termination of a signal handler started during a *pause()*, the *pause()* call will return.

RETURN VALUE

Always returns –1.

ERRORS

Pause always returns:

[EINTR] The call was interrupted.

SEE ALSO

kill(2), select(2), sigpause(2),alarm(3c)

NAME

perror, strerror – system error messages

SYNOPSIS

```
#include <stdio.h>

void perror(const char *s);

#include <string.h>

char *strerror(int errnum);
```

DESCRIPTION

perror produces a short error message on the standard error file describing the last error encountered during a call to the system from a C program. First the argument string *s* is printed, then a colon, then the message and a new-line. The argument string is often the name of the program which incurred the error. The error number is taken from the external variable **errno** (see *intro*(2)), which is set when errors occur but not cleared when non-erroneous calls are made.

strerror returns a string containing a short error message related to the argument *errnum*. On ConvexOS, this message describes errors for the values that **errno** could acquire. **errno** can be used as the argument to *strerror*.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- *strerror* is not available in the backward-compatible mode.

SEE ALSO

intro(2), psignal(3)

NAME

plot: openpl, erase, label, line, circle, arc, move, cont, point, linemod, space, closepl – graphics interface

SYNOPSIS

openpl()

erase()

label(s)
char s[];

line(x1, y1, x2, y2)

circle(x, y, r)

arc(x, y, x0, y0, x1, y1)

move(x, y)

cont(x, y)

point(x, y)

linemod(s)
char s[];

space(x0, y0, x1, y1)

closepl()

DESCRIPTION

These subroutines generate graphic output in a relatively device-independent manner. See *plot*(5) for a description of their effect. *Openpl* must be used before any of the others to open the device for writing. *Closepl* flushes the output.

String arguments to *label* and *linemod* are null-terminated and do not contain newlines.

Various flavors of these functions exist for different output devices. They are obtained by the following *ld*(1) options:

–lplot device-independent graphics stream on standard output for *plot*(1) filters
–l300 GSI 300 terminal
–l300s GSI 300S terminal
–l450 DASI 450 terminal
–l4014 Tektronix 4014 terminal

SEE ALSO

graph(1G), plot(1G), plot(5)

NAME

popen, pclose – initiate I/O to/from a process or pipe

SYNOPSIS

#include <stdio.h>

FILE *popen(command, type)
char *command, *type;

pclose(stream)
FILE *stream;

DESCRIPTION

The arguments to *popen* are pointers to null-terminated strings containing respectively a shell command line and an I/O mode, either "r" for reading or "w" for writing. It creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer that can be used (as appropriate) to write to the standard input of the command or read from its standard output.

A stream opened by *popen* should be closed by *pclose*, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type "r" command may be used as an input filter, and a type "w" as an output filter.

SEE ALSO

pipe(2), fopen(3S), fclose(3S), system(3), wait(2), sh(1)

DIAGNOSTICS

Popen returns a null pointer if files or processes cannot be created, or the shell cannot be accessed.

Pclose returns –1 if *stream* is not associated with a 'popened' command.

BUGS

Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be forestalled by careful buffer flushing, for instance, with *fflush,* see *fclose*(3).

Popen always calls *sh*, never calls *csh*.

NAME

printf, fprintf, sprintf – formatted output conversion

SYNOPSIS

#include <stdio.h>

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

int fprintf(FILE *stream, const char *format, ...);

int sprintf(char *s, const char *format, ...);

DESCRIPTION

printf places output on the standard output stream *stdout*. *fprintf* places output on the named output *stream*. *sprintf* places output in the string *s*, followed by the character **\0**.

Each of these functions converts, formats, and prints its arguments under control of the first argument. The first 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 which causes conversion and printing of the next successive argument in the argument list.

Each conversion specification is introduced by the character **%**.

After the **%**, the following appear in order:

- Zero or more *flags* in any order (described later) that modify the meaning of the conversion specification.
- An optional digit string specifying a *field width;* if the converted value has fewer characters than the field width, it will be padded with spaces on the left by default (or right, if the left adjustment flag, described later, has been given) to make up the field width.
- An optional *precision* which consists of a **.** , which serves to separate the precision from the width, followed by a decimal integer. The precision gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point character for **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum characters to be written from a string in s conversions. If only a **.** is specified, a precision of 0 is implied.
- An optional **h** before an integral conversion means that the argument is of type **short int** or **unsigned short int** although the argument will have been promoted according to normal integral promotions. An **h** before a **n** specifies that the argument is a **short int***. The optional **l** specifies that a following integral conversion corresponds to a long 32-bit integer argument. The **l** followed by a **n** specifies that the argument is a **long int *.** The optional **L** means that the following floating-point conversion specifier applies to an argument of type **long double.** The **ll** and **LL** are CONVEX extensions specifying that the following integral conversion applies to **long long int** (64-bit integer variables).
- A character (enumerated later) that specifies the type of conversion.

An asterisk may be used in place of the field width or precision. In this case, an **int** argument is supplied in the argument list. A negative field width argument is taken as a – flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are:

– The result of the conversion will be left-justified.

\+ The result of the conversion will have a plus or minus sign.

space If the first character of a signed conversion is not signed, or if the conversion results in characters, a space will be prefixed to the result. This flag is ignored if a + flag is also used.

An optional **#** specifying that the value should be converted to an "alternate form". For **o** conversions, the precision of the number is increased to force the first character of the output string to a zero. For **x(X)** conversion, a nonzero result has the string **0x(0X)** prepended to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point only appears in the results of those conversions if a digit follows the decimal point). For **g** and **G** conversions, trailing zeros are not removed from the result.

0 For all numeric conversions, leading zeros following any sign or base are used to pad the field width. If the – flag is also used, the **0** flag is ignored. For integral conversions this flag is ignored if a precision is also specified.

The conversion characters and their meanings are:

d,i The integer argument is converted to a signed decimal. The precision specifies the *minimum* number of digits to appear; if the value is not large enough to use the indicated number of digits, it is padded with leading zeros. The default precision is 1. The result of converting the value zero with a precision of zero is no characters.

o,u,x,X The argument is of type **unsigned int** and is converted to unsigned octal, decimal, or hexadecimal notation. The letters a through f are used if the **x** specifier is used and A through F are used if the **X** specifier is used. The precision specifies the *minimum* number of digits to appear; if the value is not large enough to use the indicated number of digits, it is padded with leading zeros. The default precision is 1. The result of converting the value zero with a precision of zero is no characters.

f The argument of type **double** 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 **0** and the **#** was not specified, no decimal point is printed. If a decimal point character appears, at least one digit appears before it.

e,E The double argument 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. The precision defaults to 6. If the precision is **0** and the **#** was not specified, no decimal point is printed. The exponent will always contain at least two digits. If the value is zero, the exponent is zero. Uppercase **E** causes the **e** in the exponent to be printed in uppercase.

g,G The **double** argument is printed in style **d**, in style **f**, or in style **e**. The precision specifies the the number of significant digits. If the precision is zero, it taken as 1. The style depends on the value. The style **e** or **E** will be used only if the exponent resulting from such a conversion is less that -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion. A decimal only appears if it is followed by a digit. Uppercase **G** causes the **e** in the exponent (if any) to be printed in uppercase.

c The argument is converted to an **unsigned char** and is printed.

s The argument is taken to be an array of characters and characters from the array are printed until a null character or until 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. If the precision is not specified or is greater than the size of the array, the array will contain a null character.

p The argument is a pointer to **void.** The value of the pointer is printed as a hexadecimal integer.

n The argument is a pointer to an integer into which is *written* the number of characters transmitted thus far. No argument is converted.

% Prints a %; no argument is converted.

In no case does a nonexistent or small *field width* cause truncation of a field. Padding takes place only if the specified *field width* exceeds the actual width.

In all of the floating-point output formats, the output is converted to the specified precision by rounding.

RETURNS

These routines return the number of characters transmitted or a negative value if an output error occurred.

EXAMPLES

To print a date and time in the form 'Sunday, July 3, 10:02', where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %02d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimals:

```
printf("pi = %.5f" 4*atan(1.0));
```

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- The backward-compatible *sprintf* returns a pointer the its first argument. *printf* and *fprintf* return 0 or **EOF** to indicate status.
- The +, *space*, and **0** flags are not available in the backward-compatible mode.
- **#** is not considered a flag and must appear *after* the precision in the backward-compatible mode.
- The precision specifier is not allowed with integral conversions in the backward-compatible mode.
- The **%i** conversion specifier is unavailable in the backward-compatible mode.
- The **%p** conversion specifier is unavailable in the backward-compatible mode.
- The **%n** conversion specifier is unavailable in the backward-compatible mode.

SEE ALSO

putc(3S), scanf(3S), ecvt(3) va_list(3) vprintf(3s)

BUGS/NOTES

The **%hn** and **%ln** are not implemented.

The *printf* family of routines makes use of *ecvt, gcvt,* and *fcvt.* These conversion routines use an internal buffer which may be overwritten by the *printf* family of routines when floating-point numbers are printed.

NAME

psignal, sys_siglist – system signal messages

SYNOPSIS

psignal(sig, s)
unsigned sig;
char *s;

char *sys_siglist[];

DESCRIPTION

Psignal produces a short message on the standard error file describing the indicated signal. First the argument string *s* is printed, then a colon, then the name of the signal and a new-line. Most usefully, the argument string is the name of the program which incurred the signal. The signal number should be from among those found in *<signal.h>*.

To simplify variant formatting of signal names, the vector of message strings *sys_siglist* is provided; the signal number can be used as an index in this table to get the signal name without the newline. The define NSIG defined in *<signal.h>* is the number of messages provided for in the table; it should be checked because new signals may be added to the system before they are added to the table.

SEE ALSO

sigvec(2), perror(3)

NAME

putc, putchar, fputc, putw – put character or word on a stream

SYNOPSIS

#include <stdio.h>

int putc(int c, FILE *stream);

int putchar(int c);

int fputc(int c, FILE *stream);

int putw(int w, FILE *stream);

DESCRIPTION

putc appends **c** to the named output *stream*. It returns the character written.

putchar(*c*) is defined as *putc*(*c*, **stdout**).

fputc behaves like *putc*, but is a genuine function rather than a macro.

putw appends word (that is, **int**) *w* to the output *stream*. It returns the word written. *putw* neither assumes nor causes special alignment in the file.

SEE ALSO

fopen(3S), fclose(3S), getc(3S), puts(3S), printf(3S), fread(3S)

DIAGNOSTICS

These functions return the constant **EOF** upon error. Since this is a good integer, *ferror*(3S) should be used to detect *putw* errors.

BUGS

Because it is implemented as a macro, *putc* treats a *stream* argument with side effects improperly. In particular:

```
putc(c, *f++);
```

does not work sensibly.

To get *lint* to be quiet about references to *putc* and *putchar*, one must actually use the value that these functions return; using **void** will not work. Below is an example of a reference to *putc* that will keep *lint* happy:

```
if (putc ('a', stdout) == EOF) {
        fprintf (stderr, "Could not putc 'a' to stdout.\n");
        exit (1);
}
```

Errors can occur long after the call to *putc*.

NAME

puts, fputs – put a string on a stream

SYNOPSIS

#include <stdio.h>

int puts(const char *s);

int fputs(const char *s, FILE *stream);

DESCRIPTION

puts copies the null-terminated string *s* to the standard output stream **stdout** and appends a newline character.

fputs copies the null-terminated string *s* to the named output *stream*.

Neither routine copies the terminal null character.

SEE ALSO

fopen(3S), gets(3S), putc(3S), printf(3S), ferror(3S)
fread(3S) for *fwrite*

BUGS

puts appends a newline, *fputs* does not, all in the name of backward compatibility.

NAME

putshares - write share data base file entry

SYNOPSIS

#include <shares.h>

int putshares(lp, extime)
struct lnode * lp;
unsigned long extime;

DESCRIPTION

putshares writes the share data base entry with the same uid as *lp->l_uid*.

putshares returns 0 if *lp->l_uid* is greater than MAXUID. If the write is successful, *putshares* returns the size of the structure written. *putshares* returns -1 if any error occurs.

FILES

/etc/shares	share data-base file
/usr/lib/libshare.a	the share-specific library routines

SEE ALSO

closeshares(3), getshares(3), getshput(3), openshares(3), sharesfile(3)

NOTE

The Share scheduler is an optional product; for more information, contact your CONVEX sales representative.

NAME

qsort, bsearch – quicker sort and search

SYNOPSIS

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
        int compare(const void *x, const void *y));

void bsearch(const  void *key,  const void *base,
        size_t nmemb, size_t size,
        int compare(const void *x, const void *y));
```

DESCRIPTION

qsort is an implementation of the quicker-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 width 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 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.

bsearch searches an array of *nmemb* objects of size *size* pointed to by *base* for *key*. The array is assumed sorted according to the *compare* function, which is used to search for the *key*. *bsearch* returns a pointer to the array element that matches *key* or **NULL** if a match is not found.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- *bsearch* is not available in the backward-compatible mode.

SEE ALSO

sort(1)

NAME

rand, srand – random number generator

SYNOPSIS

#include <stdlib.h>

int rand(void);

void srand(unsigned int seed);

DESCRIPTION

rand uses a non-linear additive feedback random number generator employing a default table of 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$. The period of this random number generator is very large, approximately $16*(2^{31}-1)$.

The generator is reinitialized by calling *srand* with 1 as argument. *rand* can be set to produce a different pseudo-random sequence of numbers by calling *srand* with a different argument.

BACKWARD COMPATIBILITY

When compiling with the *-pcc* flag of the **cc** command, different library routines are linked. *rand* in the backward-compatible libraries is compatible with previous versions of *rand*. It uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$.

NAME

random, srandom, initstate, setstate – better random number generator; routines for changing generators

SYNOPSIS

```
long random()

srandom(seed)
int seed;

char *initstate(seed, state, n)
unsigned seed;
char *state;
int n;

char *setstate(state)
char *state;
```

DESCRIPTION

Random uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$. The period of this random number generator is very large, approximately $16*(2^{31}-1)$.

Random/srandom have (almost) the same calling sequence and initialization properties as *rand/srand.* The difference is that *rand*(3) produces a much less random sequence -- in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by *random* are usable. For example, "random()&01" will produce a random binary value.

Unlike *srand, srandom* does not return the old seed; the reason for this is that the amount of state information used is much more than a single word. (Two other routines are provided to deal with restarting/changing random number generators). Like *rand*(3), however, *random* will by default produce a sequence of numbers that can be duplicated by calling *srandom* with *1* as the seed.

The *initstate* routine allows a state array, passed in as an argument, to be initialized for future use. The size of the state array (in bytes) is used by *initstate* to decide how sophisticated a random number generator it should use -- the more state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error). The seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. *Initstate* returns a pointer to the previous state information array.

Once a state has been initialized, the *setstate* routine provides for rapid switching between states. *Setstate returns a pointer to the* argument state array is used for further random number generation until the next call to *initstate* or *setstate.*

Once a state array has been initialized, it may be restarted at a different point either by calling *initstate* (with the desired seed, the state array, and its size) or by calling both *setstate* (with the state array) and *srandom* (with the desired seed). The advantage of calling both *setstate* and *srandom* is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2^{69}, which should be sufficient for most purposes.

AUTHOR

Earl T. Cohen

DIAGNOSTICS

If *initstate* is called with less than 8 bytes of state information, or if *setstate* detects that the state information has been garbled, error messages are printed on the standard error output.

SEE ALSO

rand(3)

BUGS

About 2/3 the speed of *rand*(3C).

NAME

rcmd, rresvport, ruserok – routines for returning a stream to a remote command

SYNOPSIS

```
rem = rcmd(ahost, inport, locuser, remuser, cmd, fd2p);
char **ahost;
u_short inport;
char *locuser, *remuser, *cmd;
int *fd2p;

s = rresvport(port);
int *port;

ruserok(rhost, superuser, ruser, luser);
char *rhost;
int superuser;
char *ruser, *luser;
```

DESCRIPTION

rcmd is a routine used by the super-user to execute a command on a remote machine using an authentication scheme based on reserved port numbers. *rresvport* is a routine which returns a descriptor to a socket with an address in the privileged port space. *ruserok* is a routine used by servers to authenticate clients requesting service with *rcmd*. All three functions are present in the same file and are used by the *rshd*(8C) server (among others).

rcmd looks up the host **ahost* using *gethostbyname*(3N), returning –1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

If the call succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

The protocol is described in detail in *rshd*(8C).

The *rresvport* routine is used to obtain a socket with a privileged address bound to it. This socket is suitable for use by *rcmd* and several other routines. Privileged Internet ports are those in the range 0 to 1023. Only the super-user is allowed to bind an address of this sort to a socket.

ruserok takes a remote host's name, as returned by a *gethostbyaddr*(3N) routine, two user names and a flag indicating whether the local user's name is that of the super-user. It then checks the files */etc/hosts.equiv* and, possibly, *.rhosts* in the local user's home directory to see if the request for service is allowed. A 0 is returned if the machine name is listed in the "hosts.equiv" file, or the host and remote user name are found in the ".rhosts" file; otherwise *ruserok* returns –1. If the *superuser* flag is 1, the checking of the "host.equiv" file is bypassed.

SEE ALSO

intro(2), hosts.equiv(5), rlogin(1C), rsh(1C), rexec(3X), rexecd(8C), rlogind(8C), rshd(8C)

DIAGNOSTICS

rcmd returns a valid socket descriptor on success. It returns –1 on error and prints a diagnostic message on the standard error.

rresvport returns a valid, bound socket descriptor on success. It returns –1 on error with the global value *errno* set according to the reason for failure. The error code EAGAIN is overloaded to mean "All network ports in use."

NOTES

rcmd is an optional product; for more information, contact your CONVEX sales representative.

NAME

rcvtir,dcvtir,ircvtr,idcvtd – IEEE/native floating point mode conversion routines.

SYNOPSIS

```
#include <math.h>

float rcvtir(x)
float x;

double dcvtid(x)
double x;

float ircvtr(x)
float x;

double idcvtd(x)
double x;
```

DESCRIPTION

rcvtir returns the value of its single precision native mode input in IEEE mode. A dirty zero (i.e. non-zero fractional part) will be returned as zero.

dcvtid returns the value of its double precision native mode input in IEEE mode. A dirty zero (i.e. non-zero fractional part) will be returned as zero.

ircvtr returns the value of its single precision IEEE mode input in native mode. A zero (dirty, or true) will be returned as zero.

idcvtd returns the value of its double precision IEEE mode input in native mode. A +/- zero (dirty, or true) will be returned as zero.

DIAGNOSTICS

rcvtir and *dcvtid* have the following boundry conditions and return values:

```
underflow -> 0
     Rop -> +Inf
```

ircvtr and *idcvtd* have the following boundry conditions and return values:

```
+/- Inf -> Rop
 +- Nan -> Rop
overflow -> Rop
```

FILES

/usr/lib/libm.a

/usr/lib/libF77.a

BUGS

There are problems with the single precision versions due to the conversion of function arguments to double precision in C. A reserved operand trap will result for *ircvtr* when negative zero (dirty or true) is passed as the argument. A call to *rcvtir* with a reserved operand as argument will result in a reserved operand trap prior to the call, rather than returning a reserved operand.

NAME

re_comp, re_exec – regular expression handler

SYNOPSIS

char *re_comp(s)
char *s;

re_exec(s)
char *s;

DESCRIPTION

Re_comp compiles a string into an internal form suitable for pattern matching. *Re_exec* checks the argument string against the last string passed to *re_comp*.

Re_comp returns 0 if the string *s* was compiled successfully; otherwise a string containing an error message is returned. If *re_comp* is passed 0 or a null string, it returns without changing the currently compiled regular expression.

Re_exec returns 1 if the string *s* matches the last compiled regular expression, 0 if the string *s* failed to match the last compiled regular expression, and –1 if the compiled regular expression was invalid (indicating an internal error).

The strings passed to both *re_comp* and *re_exec* may have trailing or embedded newline characters; they are terminated by nulls. The regular expressions recognized are described in the manual entry for *ed*(1), given the above difference.

SEE ALSO

ed(1), ex(1), grep(1)

DIAGNOSTICS

Re_exec returns –1 for an internal error.

Re_comp returns one of the following strings if an error occurs:

No previous regular expression,
Regular expression too long,
unmatched \(,
missing],
too many \(\) pairs,
unmatched \).

NAME

res_mkquery, res_send, res_init, dn_comp, dn_expand – resolver routines

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

res_mkquery(op, dname, class, type, data, datalen, newrr, buf, buflen)
int op, class, type, datalen, buflen;
char *dname, *data, *buf;
struct rrec *newrr;

res_send(msg, msglen, answer, anslen)
char *msg, *answer;
int msglen, anslen;

res_init()

dn_comp(exp_dn, comp_dn, length, dnptrs, lastdnptr)
char *exp_dn, *comp_dn, **dnptrs, **lastdnptr;
int length;

dn_expand(msg, eomorig, comp_dn, exp_dn, length)
char *msg, *eomorig, *comp_dn, exp_dn;
int length;
```

DESCRIPTION

These routines are used for making, sending and interpreting packets for use with Internet domain name servers. Global information that is used by the resolver routines is kept in the variable *_res*. Most of the values have reasonable defaults and can be ignored. Options stored in *_res.options* are defined in *resolv.h* and are as follows. Options are stored a simple bit mask containing the bitwise "or" of the options enabled.

RES_INIT
: True if the initial name server address and default domain name are initialized (i.e., *res_init* has been called).

RES_DEBUG
: Print debugging messages.

RES_AAONLY
: Accept authoritative answers only. With this option, *res_send* should continue until it finds an authoritative answer or finds an error. Currently this is not implemented.

RES_USEVC
: Use TCP connections for queries instead of UDP datagrams.

RES_STAYOPEN
: Used with RES_USEVC to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the normal mode used.

RES_IGNTC
: Unused currently (ignore truncation errors, i.e., don't retry with TCP).

RES_RECURSE
: Set the recursion-desired bit in queries. This is the default. (*res_send* does not do iterative queries and expects the name server to handle recursion.)

RES_DEFNAMES
: If set, *res_mkquery* will append the default domain name to single-component names

(those that do not contain a dot). This is the default.

RES_DNSRCH

If this option is set, the standard host lookup routine *gethostbyname*(3) will search for host names in the current domain and in parent domains; see *hostname*(7).

res_init reads the initialization file to get the default domain name and the Internet address of the initial hosts running the name server. If this line does not exist, the host running the resolver is tried. The control file */etc/use_nameserver* must exist in order to use the name server. *res_init* returns an immediate error if this file does not exist.

res_mkquery makes a standard query message and places it in *buf*. *res_mkquery* will return the size of the query or –1 if the query is larger than *buflen*. *Op* is usually QUERY but can be any of the query types defined in *nameser.h*. *dname* is the domain name. If *dname* consists of a single label and the RES_DEFNAMES flag is enabled (the default), the current domain name will be appended to *dname*. The current domain name is defined by the hostname or is specified in a system file; it can be overridden by the environment variable LOCALDOMAIN. *newrr* is currently unused but is intended for making update messages.

res_send sends a query to name servers and returns an answer. It will call *res_init* if RES_INIT is not set, send the query to the local name server, and handle timeouts and retries. The length of the message is returned, or –1 if there were errors.

dn_expand expands the compressed domain name *comp_dn* to a full domain name. Expanded names are converted to upper case. *msg* is a pointer to the beginning of the message, *exp_dn* is a pointer to a buffer of size *length* for the result. The size of compressed name is returned or -1 if there was an error.

dn_comp compresses the domain name *exp_dn* and stores it in *comp_dn*. The size of the compressed name is returned or -1 if there were errors. *length is the size of the comp_dn*. *dnptrs* is a list of pointers to previously compressed names in the current message. The first pointer points to to the beginning of the message and the list ends with NULL. *lastdnptr* is a pointer to the end of the array pointed to *dnptrs*. A side effect is to update the list of pointers for labels inserted into the message by *dn_comp* as the name is compressed. If *dnptr* is NULL, names are not compressed. If *lastdnptr* is NULL, the list of labels is not updated.

FILES

/etc/resolv.conf see resolver(5)
/etc/use_nameserver name server control file

SEE ALSO

gethostbyname(3), resolver(5), hostname(7), named(8),
RFC882, RFC883, RFC973, RFC974,
Name Server Operations Guide for BIND

NAME

rexec – return stream to a remote command

SYNOPSIS

rem = rexec(ahost, inport, user, passwd, cmd, fd2p);
char **ahost;
u_short inport;
char *user, *passwd, *cmd;
int *fd2p;

DESCRIPTION

Rexec looks up the host **ahost* using *gethostbyname*(3N), returning –1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host. If a username and password are both specified, then these are used to authenticate to the foreign host; otherwise the environment and then the user's *.netrc* file in his home directory are searched for appropriate information. If all this fails, the user is prompted for the information.

The port *inport* specifies which well-known DARPA Internet port to use for the connection; it will normally be the value returned from the call "getservbyname("exec", "tcp")" (see *getservent*(3N)). The protocol for connection is described in detail in *rexecd*(8C).

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then a auxiliary channel to a control process will be setup, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

SEE ALSO

rcmd(3X), rexecd(8C)

BUGS

There is no way to specify options to the *socket* call which *rexec* makes.

NOTES

Rexec is an optional product; for more information, contact your CONVEX sales representative.

NAME

scandir, alphasort – scan a directory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/dir.h>

scandir(dirname, namelist, select, compar)
char *dirname;
struct direct *(*namelist[]);
int (*select)();
int (*compar)();

alphasort(d1, d2)
struct direct **d1, **d2;
```

DESCRIPTION

scandir reads the directory *dirname* and builds an array of pointers to directory entries using *malloc*(3). It returns the number of entries in the array and a pointer to the array through *namelist*.

The *select* parameter is a pointer to a user supplied subroutine which is called by *scandir* to select which entries are to be included in the array. The select routine is passed a pointer to a directory entry and should return a non-zero value if the directory entry is to be included in the array. If *select* is null, then all the directory entries will be included.

The *compar* parameter is a pointer to a user supplied subroutine which is passed to *qsort*(3) to sort the completed array. If this pointer is null, the array is not sorted. *alphasort* is a routine which can be used for the *compar* parameter to sort the array alphabetically.

The memory allocated for the array can be deallocated with *free* (see *malloc*(3)) by freeing each pointer in the array and the array itself.

SEE ALSO

directory(3), malloc(3), qsort(3), dir(5)

DIAGNOSTICS

Returns –1 if the directory cannot be opened for reading or if *malloc*(3) cannot allocate enough memory to hold all the data structures.

NAME

scanf, fscanf, sscanf – formated input conversion

SYNOPSIS

```
#include <stdio.h>

int scanf(const char *format, ... );

int fscanf(FILE *stream, const char *format, ... );

int sscanf(char *s, const char *format, ... );
```

DESCRIPTION

scanf reads from the standard input stream **stdin**. *fscanf* reads from the named input *stream*. *sscanf* 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:

1. White-space characters which match optional white space in the input.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional maximum field width, an optional size indicator, and a conversion character.

The size indicator is one of the following characters: **h**, **l**, **L**, or **ll**. The **h** that precedes an integral conversion means that the argument is a pointer to a **short int** or an **unsigned short int**. The **l** that precedes an integral conversion means that the argument is a pointer to a **long int** or an **unsigned long int.** The **l** that precedes a numerical real conversion means that the argument is a pointer to a **double**. The **L** that precedes a numerical real conversion means that the argument is a pointer to a **long double**. The indicator **ll** is a CONVEX extension specifing that the following integral conversion is a pointer to a **long long int**.

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 *. 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.

The conversion character indicates the interpretation of the input field; it usually implies the type of pointer required. The following conversion characters are legal:

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

d An optionally signed decimal integer is expected; the corresponding argument should be an **int ***. The form of the integer is a sequence of decimal digits. The digits are interpeted as an integer in base 10. The form of the integer should match the input for the **strtol** function with the value 10 for the **base** argument.

i An optionally signed decimal integer is expected; the corresponding argument should be an **int ***. If the string of digits begins with a "0" followed by a string of digits, it is interpreted as an octal integer. If the string begins with "0X" or "0x" followed by a string of hexadecimal digits, it is interpreted as a hexadecimal integer. Otherwise, the sequence of decimal digits is interpreted as an integer in base 10. The form of the integer should match the input for the **strtol** function with the value 0 for the **base** argument.

o An optionally signed octal integer is expected; the corresponding argument should be an **unsigned int ***. The form of the integer should match the input for the **strtoul** function

with the value 8 for the **base** argument.

u An optionally signed decimal integer is expected; the corresponding argument should be an **unsigned int ***. The form of the integer should match the input for the **strtoul** function with the value 10 for the **base** argument.

x An optionally signed hexadecimal integer is expected; the corresponding argument should be an **unsigned int ***. The form of the integer should match the input for the **strtoul** function with the value 16 for the **base** argument.

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 "\0", which will be added automatically. The input field is terminated by a space character or a newline.

c A character is expected. The next character in the stream is assigned even if it is a white space character. The corresponding argument should be a character pointer. If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.

e,f,g 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 the **subject** argument to the **strtod** function and is an optionally signed string of digits possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or **e** followed by an optionally signed integer.

[Indicates a string that is not necessarily delimited by spaces but is composed only of characters in the *scanset.* The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the *scanset.* If the first character is a] then it is taken to be part of the scanset and another] delimits the scanset. If the first character is not circumflex (^), the input field is all characters until the first character not in the scanset. If the first character after the left bracket is ^, the input field is all characters *not* in the scanset. The corresponding argument must point to a character array.

p Matches a hexadecimal number without a **0x** prefix which is taken as a pointer to **void**. This is interpreted as an address in hexadecimal form.

n No input is consumed. The argument is a pointer to an integer through which is *written* the number of characters read thus far. Execution of this directive does not increment the assignment count.

RETURNS

These functions return **EOF** if input failure occurs before any conversion. Otherwise, the number of successfully matched and assigned input items is returned. This can be used to decide how many input items were found. These routines return when the format string is exhausted or when the input does not match the expected form.

EXAMPLES

The call

```
int i; float x; char name[50];
scanf("%d%f%s", &i, &x, name);
```

with the input line

```
25  54.32E–1  thompson
```

assigns to *i* the value 25, *x* the value 5.432, and *name* contains the null-terminated string "thompson". Or,

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

```
scanf("%2d%f% *d%[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

assigns 56 to *i*, 789.0 to *x*, skips "0123", and places the null-terminated string "56" in *name*. The next call to *getchar* returns "a".

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- The backward-compatible mode often continues assigning input items even after a conversion has failed.
- The backward-compatible mode does not completely match the input of the **strtod** function with real number conversion specifiers. For example, the input string "10e+rror" when fed to the format string "%e%c" assigns 'r' to the character argument instead of '+' which is the behavior of the conforming libraries.
- Many uppercase conversion specifiers are available in backward-compatible mode indicating **long int** arguments.
- The **L** indicator is not available in backward-compatible mode.
- The **p** conversion specifier is not available in backward-compatible mode.
- The **n** conversion specifier is not available in backward-compatible mode.
- The **i** conversion specifier is not available in backward-compatible mode.
- Form feed and vertical tab do not match the white-space directive in backward-compatible mode.
- The backward-compatible mode returns **EOF** if end of file is encountered regardless of the assignments made.

SEE ALSO

atof(3), *getc*(3S), *strtod*(3)

DIAGNOSTICS

The *scanf* functions return **EOF** on end of input, and a short count for missing or illegal data items.

BUGS

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

NAME

setbuf, setbuffer, setlinebuf, setvbuf – assign buffering to a stream

SYNOPSIS

#include <stdio.h>

void setbuf(FILE *stream, char *buf);

int setvbuf(FILE *stream, char *buf, int mode, size_t size);

int setbuffer(FILE *stream, char *buf, int size);

int setlinebuf(FILE *stream);

DESCRIPTION

The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as it is written; when it is block buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a newline is encountered or input is read from stdin. *fflush* (see *fclose*(3S)) may be used to force the block out early. Normally all files are block buffered. A buffer is obtained from *malloc*(3) upon the first *getc* or *putc*(3S) on the file. If the standard stream **stdout** refers to a terminal it is line buffered. The standard stream **stderr** is always unbuffered.

setbuf is used after a stream has been opened but before it is read or written. The character array *buf* is used as the stream buffer instead of a buffer automatically allocated. If *buf* is the constant pointer NULL, input/output will be completely unbuffered. Otherwise a buffer of size BUFSIZ is assumed.

setvbuf is used after a stream has been opened but before it is read or written. The *mode* argument determines how the stream will be buffered: **_IOFBF, _IOLBF,** or **_IONBF** to indicate full, line, or no buffering. If *buf* is not **NULL**, then *buf* must point to an array of *size* characters to be used as a buffer for the stream. *setvbuf* returns 0 if successful; nonzero if an improper mode is passed as *mode*.

setbuffer, an alternate form of *setbuf*, is used after a stream has been opened but before it is read or written. The character array *buf* whose size is determined by the *size* argument is used instead of an automatically allocated buffer. If *buf* is the constant pointer NULL, input/output will be completely unbuffered.

setlinebuf is used to change *stdout* or *stderr* from block buffered or unbuffered to line buffered. Unlike *setbuf* and *setbuffer* it can be used at any time that the stream is active.

A file can be changed from unbuffered or line buffered to block buffered by using *freopen* (see *fopen*(3S)). A file can be changed from block buffered or line buffered to unbuffered by using *freopen* followed by *setbuf* with a buffer argument of NULL.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- *setvbuf* is not available in the backward-compatible mode.

SEE ALSO

fopen(3S), getc(3S), putc(3S), malloc(3), fclose(3S), puts(3S), printf(3S), fread(3S)

BUGS

The standard error stream should be line buffered by default.

NAME

setfpmode – set floating point mode during program execution

SYNOPSIS

#include <convex/fpmode.h>

int setfpmode(value, level)
int value;
int level;

DESCRIPTION

Usually used by dual-mode programs, *setfpmode* will set the floating point mode of a program while it is executing. The valid *value*s are FPMODE_IEEE and FPMODE_NATIVE (defined in *<convex/fpmode.h>*), which turn the IEEE bit of the PSW on or off, respectively.

The second argument, *level*, is used to specify where the given change should take affect. If *level* is 0, the entire program's floating point mode will be set to *value*. Only the current (calling) routine will be modified if *level* is 1. And for a *level* greater than 1, only the *level*th ancestor routine will have its mode set to *value*.

Although the *setfpmode* routine will return a negative value if IEEE hardware is needed but not available, it is a good idea for the user to take precautions for his own sake. When setting the floating point mode to FPMODE_IEEE, care should be taken by the user program to verify that the machine can support the mode. The *getsysinfo* system call can be used to obtain this information.

SEE ALSO

getfpmode(3), getsysinfo(2)

DIAGNOSTICS

A value of -1 is returned if an error is encountered, such as an invalid mode or lack of necessary hardware.

CAVEATS

Caution must be taken if a program uses a *setjmp* before it calls *setfpmode*. A call to *setfpmode* will only change the IEEE bits of the PSW's that are saved on the runtime stack. Because one of the items saved in *jmp_buf* by *setjmp* is the PSW, a *longjmp* back to that location may cause a previous floating point mode to be restored, losing the change made by *setfpmode*. A workaround to this problem is to change the IEEE bit in the PSW of the *jmp_buf* immediately following the *setfpmode* call. The PSW is stored in *jmp_buf*[2], and the IEEE bit is defined in *<machine/psw.h>*.

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

#include <setjmp.h>

int setjmp(jmp_buf env);

void longjmp(jmp_buf env, int val);

int _setjmp(jmp_buf env);

int _longjmp(jmp_buf env, int val);

DESCRIPTION

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

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

longjmp 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. All accessible data have values as of the time *longjmp* was called.

setjmp and *longjmp* save and restore the signal mask *sigmask*(2), while *_setjmp* and *_longjmp* manipulate only the C stack and registers.

longjmp may not be used to restore the environment saved by *_setjmp*, and *_longjmp* may not be used to restore the environment saved by *setjmp*. An error condition occurs in both of these cases.

SEE ALSO

sigvec(2), sigstack(2), signal(3C)

NAME

setlimits - set limits structure

SYNOPSIS

```
#include <sys/types.h>
#include <sys/lnode.h>

int setlimits(limits)
struct lnode *limits;
```

DESCRIPTION

This library routine sets an in–core limits structure. If necessary, it also sets any group limits structures. *limits* points to an *lnode*.

The *lnode* pointed to by the argument *limits* is first examined to see if its scheduling group is *root*. If not, the *lnode* for the group is obtained (via *getshares*(3)) and passed to a recursive call to *setlimits*. Finally the original *lnode* is set with the L_SETLIM call to the *limits*(2) system call.

If the details for any group encountered cannot be found in the *limits* data-base, then the group is set to *root*.

Note that the */etc/shares* file may be left open by this routine.

FILES

/etc/shares	share data-base file
/usr/lib/libshare.a	the share-specific library routines

SEE ALSO

limits(2), closeshares(3), getshares(3), setupshares(3)

DIAGNOSTICS

As for the *limits*(2) system call.

Any error causes a -1 to be returned.

NOTE

The Share scheduler is an optional product; for more information, contact your CONVEX sales representative.

NAME

setlocale, localeconv – functions for locale manipulation

SYNOPSIS

#include <locale.h>

char *setlocale(int category, const char *locale);

struct lconv *localeconv(void);

DESCRIPTION

To solve problems faced by international users of the C programming language, ANSI introduced the notion of a *locale*. The locale is intended to describe international features that the library routines will subsequently act upon. Some of the international features are the alphabet, collation, currency formatting, date and time. The standard requires the implementation of the "C" locale, which is the minimum environment for translation with respect to locale. CONVEX supports one locale, the "C" locale.

setlocale queries or sets a portion of the program's locale. If *locale* is **NULL,** the value of the *category* for the current locale is returned. This value may subsequently be used by the *setlocale* function to restore that portion of the program's locale. To set a portion of program's locale, setlocale should be called with a supported category and locale. The categories supported are **LC_CTYPE, LC_COLLATE, LC_TIME, LC_NUMERIC,** and **LC_MONETARY.** The "C" locale is the only valid locale. Calling setlocale with **LC_ALL** for a *category* indicates that all the categories are affected. If *locale* is "" the "default" locale is specified. The user can set the default locale by setting the environment variables **LC_CTYPE, LC_COLLATE, LC_TIME, LC_NUMERIC,** or **LC_MONETARY** to a valid locale or by setting the environment variable **LANG** to a valid locale.

localeconv fills in a structure with values appropriate for formatting numeric quantities in the current locale.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. These routines are not available backward-compatible mode.

RETURNS

setlocale returns the value of the locale set, or the current locale if *locale* was **NULL**

localeconv returns a pointer to structure with the values appropriate for formatting numeric quantities. Since CONVEX only supports the "C" locale, the fields of the structure will contain the values required by the "C" locale.

NOTES

These functions are intended for use on systems implementing several locales and are provided for standard conformance.

SEE ALSO

setenv(3) strxfrm(3) mbstowcs(3)

NAME

setuid, setgid – set user and group ID

SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>

int  setuid(uid)
uid_t uid;

int  setgid(gid)
gid_t gid;
```

DESCRIPTION

If the caller's effective uid is super-user, *setuid()* sets the real user ID, effective user ID, and saved set-user-ID of the current process to the value of *uid*.

If the caller is not super-user, but *uid* is equal to the real user ID or saved set-user-ID, *setuid()* sets the effective user ID to *uid*; the real user ID remains unchanged.

If the caller's effective gid uid is super-user, *setgid()* sets the real group ID, effective group ID, and saved set-group-ID of the current process to *gid*.

If the caller is not super-user, but *gid* is equal to the real group ID or the saved set-group-ID, the *setgid()* function sets the effective group ID to *gid*; the real group ID remains unchanged.

Any supplementary group IDs of the calling process remain unchanged by calls to *setgid()* or *setuid()*.

RETURN VALUE

Zero is returned if the user (group) ID is set; –1 is returned otherwise.

ERRORS

setuid() fails if:

[EINVAL]	The value of *uid* is invalid.
[EPERM]	The caller is not super-user and *uid* does not match the real user ID or saved set-user-ID.

setgid() fails if:

[EINVAL]	The value of *gid* is invalid.
[EPERM]	The caller is not super-user and *gid* does not match the real group ID or saved set-user-ID.

BACKWARD COMPATIBILITY

seteuid, setruid, setegid, setrgid – set real or effective IDs

```
seteuid(euid)
setruid(ruid)
setegid(egid)
setrgid(rgid)
```

seteuid() (*setegid()*) sets the effective user ID (group ID) of the current process.

setruid() (*setrgid()*) sets the real user ID (group ID) of the current process.

SEE ALSO

setreuid(2), setregid(2), getuid(2), getgid(2)

NAME

setupshares - set kernel shares for a user

SYNOPSIS

setupshares(uid, efp)
int uid;
void (*efp)();

DESCRIPTION

This library routine sets up a kernel shares structure for the user represented by *uid*. *setupshares* extracts the share details for the user from the shares data-base, decays the usage figure up to the current time, and uses *setlimits*(3) to install the shares in the kernel.

If the system is out of *lnode* structures, then the structure for the default user "other" is used. If this also fails, then the structure for the super-user is used.

If there are any errors, and the second argument is non-NULL, the function will be called with a *printf*(3) format string and at most one extra argument. A non-zero result is returned for unrecoverable errors. Otherwise, *setupshares* returns **0**.

This routine is safe to use on systems where the share scheduler has not been installed or is inactive.

FILES

/etc/shares	share data-base file
/usr/lib/libshare.a	the share-specific library routines

DIAGNOSTICS

setupshares returns a non-zero result if *setlimits*(3) returns an error other than ETOOMANYU. The optional error routine is called if *setlimits*(3) returns any error, or if no shares have been allocated to the user.

SEE ALSO

getshares(3), closeshares(3), setlimits(3), share(5)

NOTE

The Share scheduler is an optional product; for more information, contact your CONVEX sales representative.

NAME

sharesfile - change name of shares file

SYNOPSIS

int sharesfile(s)
char * s;

DESCRIPTION

This routine closes the old share data base file (if open) and resets its name to the string passed. The new file must then be opened with *openshares*(3).

sharesfile always returns 0.

FILES

/etc/shares	share data-base file
/usr/lib/libshare.a	the share-specific library routines

SEE ALSO

closeshares(3), getshares(3), getshput(3), openshares(3), putshares(3)

NOTE

The Share scheduler is an optional product; for more information, contact your CONVEX sales representative.

NAME

signal, raise – simplified software signal facilities

SYNOPSIS

#include <signal.h>

(*signal(sig, func))()
void (*func)();

int raise(signo)
int signo;

DESCRIPTION

Signal is a simplified interface to the more general *sigvec*(2) facility. Programs that use *signal* in preference to *sigvec* are more likely to be portable to all UNIX systems.

raise sends the signal *signo* to the executing process.

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background (see *tty*(4)). Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals, the *signal* call allows signals either to be ignored or to cause an interrupt to a specified location. The following is a list of all signals with names as in the include file <*signal.h*>:

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction
SIGTRAP	5*	trace trap
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGURG	16•	urgent condition present on socket
SIGSTOP	17†	stop (cannot be caught or ignored)
SIGTSTP	18†	stop signal generated from keyboard
SIGCONT	19•	continue after stop
SIGCHLD	20•	child status has changed
SIGTTIN	21†	background read attempted from control terminal
SIGTTOU	22†	background write attempted to control terminal
SIGIO	23•	i/o is possible on a descriptor (see *fcntl*(2))
SIGXCPU	24	cpu time limit exceeded (see *setrlimit*(2))
SIGXFSZ	25	file size limit exceeded (see *setrlimit*(2))
SIGVTALRM	26	virtual time alarm (see *setitimer*(2))
SIGPROF	27	profiling timer alarm (see *setitimer*(2))
SIGWINCH	28•	window size change
SIGLOST	29*	resource lost (see *lockd*(8C))

SIGUSR1	30	user-defined signal 1
SIGUSR2	31	user-defined signal 2

The starred signals in the list above cause a core image if not caught or ignored.

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination (with a core image for starred signals) except for signals marked with • or †. Signals marked with • are discarded if the action is SIG_DFL; signals marked with † cause the process to stop. If *func* is SIG_IGN the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs further occurrences of the signal are automatically blocked and *func* is called.

A return from the function unblocks the handled signal and continues the process at the point it was interrupted. **Unlike previous signal facilities, the handler *func* remains installed after a signal has been delivered.**

If a caught signal occurs during certain system calls, causing the call to terminate prematurely, the call is automatically restarted. In particular this can occur during a *read*(2) or *write*(2) on a slow device (such as a terminal; but not a file) and during a *wait*(2).

The value of *signal* is the previous (or initial) value of *func* for the particular signal.

After a *fork*(2) or *vfork*(2) the child inherits all signals. *execve*(2) resets all caught signals to the default action; ignored signals remain ignored.

NOTES

The signal handling routine can be declared:

```
handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here *sig* is the signal number, into which the hardware faults and traps are mapped as defined below. *Code* is a parameter which further defines the type of hardware exception which occurred. *Scp* is a pointer to the *sigcontext* structure (defined in <*signal.h*>), used to restore the context from before the signal.

The following defines the mapping of hardware traps to signals and codes. All of these symbols are defined in <*signal.h*>:

Hardware condition	Signal	Code
Arithmetic traps:		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by zero	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Reserved Operand trap	SIGFPE	FPE_RESOP_TRAP
Segmentation Violations:		
Read access violation	SIGSEGV	SEG_READ_TRAP
Write access violation	SIGSEGV	SEG_WRITE_TRAP
Execute access violation	SIGSEGV	SEG_EXEC_TRAP
Invalid segment	SIGSEGV	SEG_INVSDR_TRAP
Invalid page table page	SIGSEGV	SEG_INVPTP_TRAP
Invalid memory reference	SIGSEGV	SEG_INVDATA_TRAP
I/O access violation	SIGSEGV	SEG_IOACC_TRAP
Ring Violations:		
Inward address reference	SIGBUS	BUS_INWADDR_TRAP
Outward ring call	SIGBUS	BUS_OUTCALL_TRAP

Inward ring return	SIGBUS	BUS_INWRTN_TRAP
Invalid syscall gate	SIGBUS	BUS_INVGATE_TRAP
Invalid return frame length	SIGBUS	BUS_INVFRL_TRAP
Illegal instruction:		
Error exit instruction	SIGILL	ILL_ERRXIT_TRAP
Privileged instruction	SIGILL	ILL_PRIVIN_TRAP
Undefined op code	SIGILL	ILL_UNDFOP_TRAP
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	

None of the above exceptions can be ignored using SIG_IGN because of the of pipelined nature of the CONVEX architecture. Indeterminate results will occur if one attempts ignore or return from a hardware-generated exception.

RETURN VALUE

signal returns the previous action on a successful call. Otherwise, –1 is returned and *errno* is set to indicate the error. *raise* returns 0 if successful, otherwise *raise* returns non-zero.

ERRORS

signal will fail and no action will take place if one of the following occur:

[EINVAL] *sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

[EINVAL] An attempt is made to ignore SIGCONT (by default SIGCONT is ignored).

WARNINGS

Execution of this or related routines in parallel may yield unexpected results.

raise is only available in the ANSI conforming modes.

SEE ALSO

kill(1), kill(2), pattach(2), sigvec(2), sigblock(2), sigsetmask(2), sigpause(2), sigstack(2), setjmp(3), tty(4)

NAME

sigprocmask – block or unblock signals

SYNOPSIS

#include <signal.h>

int sigprocmask(how, set, oset)
int mask;
sigset_t *set,*oset;

DESCRIPTION

sigprocmask() is used to examine or change the calling process's signal mask. If the argument *set* is not NULL, it points to a set of signals to be used in changing the currently blocked set.

Values for *how* are

Name	Description
SIG_BLOCK	Union of current signal mask and the signal set pointed to by *set.*
SIG_UNBLOCK	Intersection of current signal mask and the complement of the signal set pointed to by *set.*
SIG_SETMASK	Resulting signal mask is the set pointed to by *set*

If *oset* is not **NULL**, the previous mask is returned in the location pointed to by *oset.*

It is not possible to block SIGKILL or SIGSTOP; this restriction is silently imposed by the system.

The call will block if any other thread in the process is currently servicing a signal contained in mask. Execution of the call will continue whenever the thread in the signal handler returns or enables the signal via the *sigprocmask()* function call.

RETURN VALUE

On success, zero is returned. On error, –1 is returned and *errno* is set to indicated the cause of failure.

ERRORS

sigprocmask() will fail if one or more of the following is true:

[EINVAL]	The value of *how* is not one of the defined values.
[EINTR]	The calling process is multi-threaded, and the call was interrupted while waiting for another thread to complete processing of a signal in *set.*

BACKWARD COMPATIBILITY

The *sigprocmask()* function replaces *sigblock()* and *sigsetmask().*

In previous releases of the operating system, it was also impossible to block SIGCONT. This is now permitted; the effect is to defer calling any handler installed for SIGCONT until SIGCONT is unblocked; when a SIGCONT is received, however, a stopped process is continued regardless of whether SIGCONT is blocked.

SEE ALSO

kill(2), sigaction(2), sigsuspend(2)

NAME

sigsetjmp, siglongjmp – non-local goto with signal state preservation

SYNOPSIS

```
#include <setjmp.h>

int sigsetjmp(env,savemask)
sigjmp_buf env;
int savemask;

void siglongjmp(env,val)
sigjmp_buf env;
int val;
```

DESCRIPTION

The *sigsetjmp()* macro complies with the definition of the *setjmp()* macro in the C standard. If the value of the *savemask* argument is not zero, the *sigsetjmp()* function shall also save the process's current signal mask (see **<signal.h>**) as part of the calling environment.

The *siglongjmp()* function complies with the definition of the *longjmp()* function n the C Standard. If and only if the *env* argument was initialized by a call to the *sigsetjmp()* function with a non-zero *savemask* argument, the *siglongjmp()* function restores the saved signal mask.

SEE ALSO

sigaction(), sigprocmask(), sigsuspend()

NAME

sigemptyset, sigfillset, sigaddset, sigdelset, sigismember – manipulate signal sets

SYNOPSIS

```
#include <signal.h>

int sigemptyset(set)
sigset_t *set;

int sigfillset(set)
sigset_t *set;

int sigaddset(set, signo)
sigset_t *set;
int signo;

int sigdelset(set,signo)
sigset_t *set;
int signo;

int sigismember(set,signo)
sigset_t *set;
int signo;
```

DESCRIPTION

The *sigsetops* primitives manipulate sets of signals in a portable fashion.

sigemptyset() initializes the signal set pointed to by the argument *set* such that all signals are excluded.

sigfillset() initializes the signal set pointed to by the argument *set* such that all signals are included.

Either *sigfillset()* or *sigemptyset()* must be used to initialize a signal set object.

sigaddset() and *sigdelset()* respectively add to and delete from *set* the signal *signo*.

sigismember() tests whether the indicated signal *signo* is a member of *set*.

NAME

sigsuspend – wait for signal

SYNOPSIS

```
#include <signal.h>

int sigsuspend(sigmask)
sigset_t *sigmask;
```

DESCRIPTION

sigsuspend() replaces the caller's signal mask with the set of signals pointed to by the *sigmask* argument. Execution is then suspended until the deliver of a signal whose action is either to execute a signal handler or to terminate the process. If the action is to terminate the process, the *sigsuspend()* function does not return. If the action is to handle the function, *sigsuspend()* returns with the signal mask restore to the set that existed prior to the call for processes that do not specify the _SA_PARALLEL flag in the *sa_flags* member of the *sigaction* structure used to install the handler.

When calling *sigsuspend()*, the *sigmask* argument is usually the empty set to indicate that no signals are now to be blocked.

sigsuspend() always terminates by being interrupted and returns EINTR. The call will block if any other thread in the process is currently running a signal handler for any signal included in sigmask. Execution of the call will continue whenever the thread in the signal handler returns or makes a *sigsetmask()* call to re-enable the signal.

In normal usage, signals are blocked using *sigprocmask*(3) to begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using *sigsuspend()* with the mask returned by *sigprocmask*.

NOTES

sigsuspend() replaces *sigpause(2)*.

SEE ALSO

sigprocmask(3), sigaction(2), kill(2)

NAME

sin, cos, tan, asin, acos, atan, atan2, sinf, cosf, tanf, asinf, acosf, atanf, atan2f, ssin, scos, stan, sasin, sacos, satan, satan2 – trigonometric functions

SYNOPSIS

```
#include <math.h>
double sin(double x);
double cos(double x);
double tan(double x);
double asin(double x);
double acos(double x);
double atan(double x);
double atan2(double x, double y);
float sinf(float x);
float cosf(float x);
float tanf(float x);
float asinf(float x);
float acosf(float x);
float atanf(float x);
float atan2f(float x, float y);
```

DESCRIPTION

sin, cos, and *tan* return double-precision trigonometric functions of radian arguments; *ssin, scos,* and *stan* return single-precision trigonometric functions of radian arguments.

asin returns the double-precision arc sine in the range $-\pi/2$ to $\pi/2$; *asinf* returns the single-precision arc sine.

acos returns the double-precision arc cosine in the range 0 to π; *acosf* returns the single-precision arc cosine.

atan returns the double-precision arc tangent of x in the range $-\pi/2$ to $\pi/2$; *atanf* returns the single-precision arc tangent of x.

atan2 returns the double-precision arc tangent of x/y in the range $-\pi$ to π; *atan2f* returns the single-precision arc tangent of x/y.

DIAGNOSTICS

On range or domain error, **errno** is set to **ERANGE** or **EDOM.** Functions that could result in a domain error are *acos, asin, atan2, cos, sin, acosf, asinf, atan2f, cosf,* and *sinf.* Functions that could result in a range error are *sinh, cosh, sinhf,* and *coshf.* On overflow, **HUGE_VAL** of the same sign is returned.

The value of the sine and cosine functions for very large arguments is unmeaningful. This is related to the accuracy of the argument reduction. For single precision, the maximum value is $\pi * 2^{23}$. For double precision, the maximum value is $\pi * 2^{52}$. If the argument exceeds the maximum, it is replaced with zero and evaluation continues. The arc sine and arc cosine functions are undefined for values greater than 1. Arguments greater than 1 are replaced with 1 and evaluation continues. The arc tangent functions with two arguments are undefined when both arguments are 0. In this case, $\pi/2$ is returned.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- The names of the floating-point routines are *ssin, scos, stan, sasin, sacos, satan,* and *satan2* in the backward-compatible libraries.
- **errno** may be set to the folloing values in the **MTH_LRG_SIN, MTH_LRG_COS, MTH_UNDEF_ASIN, MTH_UNDEF_ACOS** or **MTH_UNDEF_ATAN2** in the backward-compatible libraries. In addition to setting **errno** , error messages are printed on domain and range error.

NAME

sinh, cosh, tanh, sinhf, coshf, tanhf, ssinh, scosh, stanh – hyperbolic functions

SYNOPSIS

```
#include <math.h>
double sinh(double x);
double cosh(double x);
double tanh(double x);
float sinhf(float x);
float coshf(float x);
float tanhf(float x);
```

DESCRIPTION

These functions compute the designated hyperbolic functions for real arguments.

DIAGNOSTICS

The possible errors and corresponding return values are:

[MTH_OVF_SINH], [MTH_OVF_COSH]

Evaluation of hyperbolic sine or cosine would lead to overflow. The maximum $|x|$ for which these functions can be computed is approximately ln(xmax+ln(2)), where xmax is the largest floating-point value. If the argument exceeds the maximum, then the largest-floating point value of the appropriate sign is returned.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- The names of the floating-point routines are *ssinh, scosh,* and *stanh.*

NAME

sleep – suspend execution for interval

SYNOPSIS

unsigned sleep(seconds)
unsigned seconds;

DESCRIPTION

The *sleep()* function causes the current process to be suspended from execution until either the number of real time seconds specified by the argument *seconds* have elapsed or a signal is delivered to the calling process and its action is to an invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested due to the scheduling of other activity by the system.

The routine is implemented by setting an interval timer and pausing until it occurs. The previous state of this timer is saved and restored. If the sleep time exceeds the time to the expiration of the previous timer, the process sleeps only until the signal would have occurred, and the signal is sent 1 second later.

The sleep routine uses the SIGALRM signal, thus temporarily redefining the existing SIGALRM signal handler.

SEE ALSO

getitimer(2), sigpause(2)

BUGS

An interface with finer resolution is needed.

WARNINGS

Execution of this routine in parallel may produce unexpected results.

NAME

stdarg.h, va_list, va_start, va_arg, va_end – variable argument list

SYNOPSIS

#include <stdarg.h>

va_list argList;
va_start(va_list argList, <last parameter>)
va_arg(va_list argList, <type>)
va_end(va_list argList)

DESCRIPTION

This set of macros provides a means of writing portable functions that accept variable argument lists (e.g., printf). Routines having variable argument lists that do not use stdarg are not portable since different machines use different argument passing conventions. (The older varargs(3) mechanism provides a slightly less portable method of writing such routines. It is available on many Unix systems but is not part of standard C).

The stdarg variable argument list system is for use with functions with prototypes using the "..." syntax to indicate that varying numbers of actual parameters may be present. There must be at least one named parameter.

va_list is a type used to declare a variable which describes the current state of the variable argument list. To use the stdarg.h system you must declare a variable of this type and initialize it with a call to **va_start**.

va_start(*argList,lastParam*) is called to initialize *argList*. The second parameter to va_start must be the last named parameter in the function's prototype.

va_arg(*argList, type*) will return the next argument in the list pointed to by *argList*. The second argument to va_arg is the type the argument is expected to be. The type may not be **short**, **char**, or **float**, or any signed, unsigned or qualified version of those types. Typically the routine determines what type the argument is based on the preceding arguments (e.g., printf's format string).

va_end(*argList*) is used to indicate that argument traversal is complete.

Multiple traversals, each bracketed by **va_start** and **va_end,** are possible, and the same va_list type variable may be reused.

The address of a **va_list** type variable may be passed to another function and **va_arg** used on the variable in that function. The **va_list** type variable must first be initialized by a call to **va_start** in the function whose parameter list is being traversed.

The stdarg macros do not check that the number of calls to va_start matches the number of arguments pushed. The routine using stdarg.h and its caller must agree on a convention for determining how many arguments there are. For example, *execl* expects a 0 argument to signal the end of the list while the format string to *printf* indicates how many arguments there should be.

EXAMPLE

```
#include <stdarg.h>
execl(char * name, ...)
{
        va_list ap;
        char *args[100];
        int argno = 0;

        va_start(ap, name);
        while (args[argno++] = va_arg(ap, char *))
                ;
        va_end(ap);
```

```
        return execv(name, args);
}
```

NAME

stdarg.h, stdarg, va_list, va_start, va_arg, va_end – variable argument list

SYNOPSIS

#include <stdarg.h>

va_list argList;
va_start(va_list argList, <last parameter>)
va_arg(va_list argList, <type>)
va_end(va_list argList)

DESCRIPTION

This set of macros provides a means of writing portable functions that accept variable argument lists (e.g., printf). Routines having variable argument lists that do not use stdarg are not portable since different machines use different argument passing conventions. (The older varargs(3) mechanism provides a slightly less portable method of writing such routines. It is available on many Unix systems but is not part of standard C).

The stdarg variable argument list system is for use with functions with prototypes using the "..." syntax to indicate that varying numbers of actual parameters may be present. There must be at least one named parameter.

va_list is a type used to declare a variable which describes the current state of the variable argument list. To use the stdarg.h system you must declare a variable of this type and initialize it with a call to **va_start.**

va_start(*argList,lastParam*) is called to initialize *argList.* The second parameter to va_start must be the last named parameter in the function's prototype.

va_arg(*argList, type*) will return the next argument in the list pointed to by *argList.* The second argument to va_arg is the type the argument is expected to be. The type may not be **short**, **char**, or **float**, or any signed, unsigned or qualified version of those types. Typically the routine determines what type the argument is based on the preceding arguments (e.g., printf's format string).

va_end(*argList*) is used to indicate that argument traversal is complete.

Multiple traversals, each bracketed by **va_start** and **va_end,** are possible, and the same va_list type variable may be reused.

The address of a **va_list** type variable may be passed to another function and **va_arg** used on the variable in that function. The **va_list** type variable must first be initialized by a call to **va_start** in the function whose parameter list is being traversed.

The stdarg macros do not check that the number of calls to va_start matches the number of arguments pushed. The routine using stdarg.h and its caller must agree on a convention for determining how many arguments there are. For example, *execl* expects a 0 argument to signal the end of the list while the format string to *printf* indicates how many arguments there should be.

EXAMPLE

```
#include <stdarg.h>
execl(char * name, ...)
{
        va_list ap;
        char *args[100];
        int argno = 0;

        va_start(ap, name);
        while (args[argno++] = va_arg(ap, char *))
                ;
        va_end(ap);
```

```
        return execv(name, args);
}
```

NAME

stddef.h, NULL, offsetof, ptr_diff_t, size_t, wchar_t – header file contains standard definitions

SYNOPSIS

#include <stddef.h>

NULL;

offsetof(type, member-designator);

ptr_diff_t;

size_t;

wchar_t;

DESCRIPTION

stddef.h contains standard definitions used by the library functions.

ptr_diff_t is the type of the result of subtracting two pointers. **size_t** is the type of the result of the **sizeof** operator.

wchar_t is the type that can hold the largest character set implemented. Currently, CONVEX supports no extended character set but **wchar_t** is provided for standard conformance.

NULL is a pointer constant having special meaning to many library functions.

offsetof expands to **size_t** which is the offset in bytes of *member-designator* from the beginning of its structure. The structure must be of type *type*.

NAME

strlen, index, rindex – string operations

SYNOPSIS

#include <string.h>

size_t strlen(s)
const char *s;

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

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

DESCRIPTION

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

strlen returns the number of characters in *s*, which must be terminated by a null character.

index returns a pointer to the first occurrence of character *c* in string *s*, or zero if *c* does not occur in the string.

rindex returns a pointer to the last occurrence of character *c* in string *s*, or zero if *c* does not occur in the string.

NOTES

rindex and *index* are available only in the extended mode and backward-compatible mode of CONVEX C.

SEE ALSO

stringcpy(3), stringcmp(3), stringsearch(3), bzero(3).

NAME

strcat, strncat – string concatenation functions

SYNOPSIS

#include <string.h>

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

char *strncat(char *s1, const char *s2, size_t n);

DESCRIPTION

strcat appends a copy of string *s2* to the end of string *s1*. *strncat* copies at most *n* characters. The target may not be null-terminated if the length of *s2* is *n* or more. Both return a pointer to the result.

SEE ALSO

stringcpy(3), stringcmp(3), stringsearch(3), bzero(3)

NAME

memcmp, strcmp, strcoll, strncmp, strxfrm – string comparison functions

SYNOPSIS

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
size_t strxfrm(char *s1, const char *s2, size_t n);
```

DESCRIPTION

memcmp compares characters pointed to by *s1* to those pointed to by *s2*.

strcmp compares the string pointed to by *s1* to the string pointed to by *s2*.

strcoll also compares the string pointed to by *s1* to the string pointed to by *s2*. The string is interpreted as appropriate to the **LC_COLLATE** category of the current locale. Currently, CONVEX only supports the "C" locale.

strncmp compares not more than *n* characters from the strings pointed to by *s1* and *s2*.

strxfrm transforms the string pointed to by *s2* into the string pointed to by *s1*. If two strings are transformed, the comparison by the *strcoll* function on the original strings is the same as the comparsion by the *strcmp* function on the transformed strings. Not more than *n* characters are placed in *s1*.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- *memcmp* is not available in the backward-compatible mode.
- *strcoll* is not available in the backward-compatible mode.
- *strxfrm* is not available in the backward-compatible mode.

RETURNS

The *memcmp, strcmp, strcoll,* and *strncmp* return an integer greater than, equal to, or less than zero according to the results of the comparison. *strxfrm* returns the length of the transformed string (not counting the **NULL** terminator).

NOTES

strcoll and *strxfrm* are provided for standard conformance. They are intended for use in environments with multiple locales.

SEE ALSO

stringcat(3), stringcpy(3), stringsearch(3), bzero(3), setlocale(3)

NAME

memcpy, memmove, memset, strcpy, strncpy – string copy functions

SYNOPSIS

#include <string.h>

void *memcpy(void *s1, const void *s2, size_t n);

void *memmove(void *s1, const void *s2, size_t n);

void *memset(void *s, int c, size_t n);

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

char *strncpy(char *s1, const char *s2, size_t n);

DESCRIPTION

memmove copies n characters from *s2* to *s1*. If the objects overlap the behavior is undefined.

memcpy copies *n* characters from *s2* to *s1*. The copying is done as if it were copied first to a temporary block so that overlapping of strings is permitted.

memset sets *n* copies of *c* into *s*.

strcpy copies string *s2* to *s1*, stopping after the null character has been moved. *strncpy* copies exactly *n* characters, truncating or null-padding *s2;* the target may not be null-terminated if the length of *s2* is *n* or more. Both *strncpy* and *strcpy* return *s1*.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following ways:

- *memcpy* is not available in the backward-compatible mode.
- *memmove* is not available in the backward-compatible mode.
- *memset* is not available in the backward-compatible mode.

SEE ALSO

stringcat(3), stringcmp(3), stringsearch(3), bzero(3)

NAME

memchr, strchr, strrchr, strcspn, strpbrk, strstr, strspn, strtok, index, rindex – string search functions

SYNOPSIS

```
#include <string.h>

void *memchr(const void *s, int c, size_t n);

char *strchr(const char *s, int c);

size_t strcspn(const char *s1, const char *s2);

char *strpbrk(const char *s1, const char *s2);

char *strrchr(const char *s, int c);

site_t strspn(const char *s1, const char *s2);

char *strstr(const char *s1, const char *s2);

char *strtok(char *s1, const char *s2);

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

char *rindex(s, c)
char *s, c;
```

DESCRIPTION

memchr locates the first occurrence of *c* (converted to an *unsigned char*) in the initial *n* characters (each interpreted as *unsigned char*) of the object pointed to by *s*.

strchr locates the first occurrence of *c* (converted to a *char*) in the string pointed to by *s*. The terminating null character is considered to be part of the string.

strcspn computes the length of the maximum initial segment of the string pointed to by *s1* which consists entirely of characters **not** from the string pointed to by *s2*.

strpbrk locates the first occurrence in the string pointed to by *s1* of any character pointed to by *s2*.

strrchr locates the last occurrence of *c* (converted to *unsigned char*) in the string pointed to by *s*. The terminating null character is considered to be part of the string.

strspn computes the length of the maximum initial segment of the string pointed to by *s1* which consists entire of character from the string pointed to by *s2*.

strstr locates the first occurrence in the string pointed to by *s1* of the sequence of characters (excluding the terminating null character) in the string pointed to by *s2*.

strtok breaks a string, *s1*, into a sequence of tokens, delimited by the characters contained in the string *s2*.

A sequence of calls to the *strtok* function breaks the string pointed to by *s1* into a sequence of tokens, each of which is delimited by a character from the string pointed to by *s2*. The first call in the sequence has *s1* as its first argument, and is followed by by calls with a null pointer as their first argument. The separator string pointed to by *s2* my be different from call to call.

The first call in the sequence searches the string pointed to by *s1* for the first character that is **not** contained in the current separator string pointed to by *s2*. If no such character is found, then there are no tokens in the string pointed to by *s1* and the *strtok* function returns a null pointer. If such a character is found, it is the start of the first token.

The *strtok* function then searches from there for a character that **is** contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by *s1*, and subsequent searched for a token will return a null pointer. If such a

character is found, it is overwritten by a null character, which terminates the current token. The *strtok* function saves a pointer to the following character, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The *index* function locates the first occurrence of *c* in the string pointed to by *s*.

The *rindex* function locates the last occurrence of *c* in the string pointed to by *s*.

RETURNS

memchr returns a pointer to the located character,or a null pointer if the character does not occur in the object.

strchr returns a pointer to the located character, or a null pointer if the character does not occur in the string.

strcspn returns the length of the segment.

strpbrk returns a pointer to the character, or a null pointer if no character from *s2* occurs in *s1*.

strrchr returns a pointer to the character, or a null pointer if *c* does not occur in the string.

strspn returns the length of the segment.

strstr returns a pointer to the located string, or a null pointer if the string is not found. If *s2* points to a string with zero length, the function returns *s1*.

strtok returns a pointer to the first character of a token, or a null pointer if there is no token. *index* returns a pointer to the located character, or zero if the character does not occur in the string. *rindex* returns a pointer to the located character, or zero if the character does not occur in the string.

BACKWARD COMPATIBILITY

Only *index* and *rindex* may be used in the backward compatible mode. These functions are also available in the extended mode the the compiler.

SEE ALSO

stringcmp(3), stringcpy(3), stringcat(3), stringlen(3)

NAME

strtod, strtol, strtoul – string to numeric value conversion routines

SYNOPSIS

#include <stdlib.h>

double strtod(const char *nptr, char **endptr);

long int *strtol(const char *nptr, char **endptr, int base);

unsigned long int *strtoul(const char *nptr, char **endpt, int base);

DESCRIPTION

strtod converts the initial portion of the string in *nptr* to a double. Whitespaces are skipped. The remainder of the string is decomposed into a subject string and a final string containing unrecognized characters. The subject sequence is an optional plus or minus, then a nonempty digit sequence optionally containing a decimal point, followed by an optional exponent. An exponent is an **e** or an **E**, then an optional plus or minus, followed by a nonempty digit sequence. If *endptr* is not **NULL**, a pointer to the final sequence is placed in **endptr*. No floating point suffix is recognized as part of the subject sequence.

strtol converts the initial portion of the string in *nptr* to a long integer. Whitespaces are skipped. The remainder of the string is decomposed into a subject string and a final string containing unrecognized characters. If the base is zero, the subject sequence optionally begins with a plus or minus. Next is a zero optionally followed by a digit sequence indicating an octal number, or a **0x** (or **0X**) followed by a digit sequence indicating a hexadecimal number, or any other digit sequence indicating a decimal number. If the value of **base** is between 2 and 36 the subject sequence is a string represented by digits and letters. The letters **a** (or **A**) through **z** (or **Z**) are ascribed values 10 to 35. If the base is 16 an optional **0x** or **0X** may follow the sign and precede the letters and digits. If *endptr* is not **NULL**, a pointer to the final sequence is placed in **endptr*. No integer suffix is recognized as part of the subject sequence.

strtoul is the same as *strtol* except that the string is interpreted as an unsigned long integer.

RETURNS

strtod returns the value of the subject string, converted to a double. If the subject string is empty or in an unexpected form, zero is returned. If the subject string represents a number outside the representable range of double, plus or minus **HUGE_VAL** is returned. If the correct value would cause underflow, zero is returned.

strtol returns the value of the subject string, converted to a long integer. If the subject string is empty or in an unexpected form, zero is returned. If the subject string represents a number outside the representable range of double, **LONG_MIN** or **LONG_MAX** is returned.

strtoul returns the the value of the subject string, converted to a long integer. If the subject string is empty or in an unexpected form, zero is returned. If the subject string represents a number outside the representable range of double, **ULONG_MAX** is returned.

DIAGNOSTICS

On range error, **errno** is set to **ERANGE**.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. These routines are not available backward-compatible mode.

SEE ALSO

atof(3), atoi(3), atol(3), atoll(3)

NAME

stty, gtty – set and get terminal state (defunct)

SYNOPSIS

```
#include <sgtty.h>

stty(fd, buf)
int fd;
struct sgttyb *buf;

gtty(fd, buf)
int fd;
struct sgttyb *buf;
```

DESCRIPTION

This interface is obsoleted by ioctl(2).

Stty sets the state of the terminal associated with *fd*. *Gtty* retrieves the state of the terminal associated with *fd*. To set the state of a terminal the call must have write permission.

The *stty* call is actually "ioctl(fd, TIOCSETP, buf)", while the *gtty* call is "ioctl(fd, TIOCGETP, buf)". See *ioctl*(2) and *tty*(4) for an explanation.

DIAGNOSTICS

If the call is successful 0 is returned, otherwise –1 is returned and the global variable *errno* contains the reason for the failure.

SEE ALSO

ioctl(2), tty(4)

NAME

swab – swap bytes

SYNOPSIS

swab(from, to, nbytes)
char *from, *to;

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the position pointed to by *to,* exchanging adjacent even and odd bytes. It is useful for carrying binary data between machines that use different byte ordering, such as CONVEX to VAX transfers. *Nbytes* should be even.

NAME

sysconf – get configurable system variables

SYNOPSIS

#include <limits.h>
#include <unistd.h>

long sysconf(name)
int name;

DESCRIPTION

The *sysconf()* function provides values of configurable system variables or options. The *name* argument represents the value to be queried. The permissible values of *name* are found in <**unistd.h**>. The values that may be interrogated follow.

_SC_ARG_MAX Maximum length of arguments and environment for the *exec* functions.

_SC_CHILD_MAX
Maximum number of simultaneous processes per real user id.

_SC_CLK_TCK Number of intervals per second, used to express the value in type *clock_t*.

_SC_NGROUPS_MAX
Maximum number of simultaneous supplementary group IDs per process.

_SC_OPEN_MAX
Maximum number of files that a process may have open at one time.

_SC_JOB_CONTROL
Job control is supported.

_SC_SAVED_IDS
Each process has a saved set-user-ID and a saved set-group-ID.

_SC_VERSION Corresponds to the revision of IEEE Std. P1003.1-1988 supported.

RETURN VALUE

Upon successful completion, the value of the requested *name* is returned. If the requested *name* has no limit, –1 is returned and *errno* is left unchanged. On error, –1 is returned and *errno* is set to indicate the error.

ERRORS

sysconf() will fail if:

[EINVAL] The value of *name* is invalid.

NOTES

The name of CLK_TCK is debatable; X3J11 seems to have changed in favor of CLOCKS_PER_SEC. One might claim that by inference, _SC_CLK_TCK should be renamed _SC_CLOCKS_PER_SEC.

NAME

syslog, openlog, closelog, setlogmask – control system log

SYNOPSIS

#include <syslog.h>

openlog(ident, logopt, facility)
char *ident;

syslog(priority, message, parameters ...)
char *message;

closelog()

setlogmask(maskpri)

DESCRIPTION

Syslog arranges to write *message* onto the system log maintained by *syslogd*(8). The message is tagged with *priority*. The message looks like a *printf*(3) string except that **%m** is replaced by the current error message (collected from *errno*). A trailing newline is added if needed. This message will be read by *syslogd*(8) and written to the system console, log files, or forwarded to *syslogd* on another host as appropriate.

Priorities are encoded as a *facility* and a *level*. The facility describes the part of the system generating the message. The level is selected from an ordered list:

LOG_EMERG	A panic condition. This is normally broadcast to all users.
LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system database.
LOG_CRIT	Critical conditions, e.g., hard device errors.
LOG_ERR	Errors.
LOG_WARNING	Warning messages.
LOG_NOTICE	Conditions that are not error conditions, but should possibly be handled specially.
LOG_INFO	Informational messages.
LOG_DEBUG	Messages that contain information normally of use only when debugging a program.

If *syslog* cannot pass the message to *syslogd*, it will attempt to write the message on */dev/console* if the LOG_CONS option is set (see below).

If special processing is needed, *openlog* can be called to initialize the log file. The parameter *ident* is a string that is prepended to every message. *Logopt* is a bit field indicating logging options. Current values for *logopt* are:

LOG_PID	log the process id with each message: useful for identifying instantiations of daemons.
LOG_CONS	Force writing messages to the console if unable to send it to *syslogd*. This option is safe to use in daemon processes that have no controlling terminal since *syslog* will fork before opening the console.
LOG_NDELAY	Open the connection to *syslogd* immediately. Normally the open is delayed until the first message is logged. Useful for programs that need to manage the order in which file descriptors are allocated.
LOG_NOWAIT	Don't wait for children forked to log messages on the console. This option should be used by processes that enable notification of child termination via SIGCHLD, as *syslog* may otherwise block waiting for a child whose exit

status has already been collected.

The *facility* parameter encodes a default facility to be assigned to all messages that do not have an explicit facility encoded:

LOG_KERN	Messages generated by the kernel. These cannot be generated by any user processes.
LOG_USER	Messages generated by random user processes. This is the default facility identifier if none is specified.
LOG_MAIL	The mail system.
LOG_DAEMON	System daemons, such as *ftpd*(8), *routed*(8), etc.
LOG_AUTH	The authorization system: *login*(1), *su*(1), *getty*(8), etc.
LOG_LPR	The line printer spooling system: *lpr*(1), *lpc*(8), *lpd*(8), etc.
LOG_LOCAL0	Reserved for local use. Similarly for LOG_LOCAL1 through LOG_LOCAL7.

Closelog can be used to close the log file.

Setlogmask sets the log priority mask to *maskpri* and returns the previous mask. Calls to *syslog* with a priority not set in *maskpri* are rejected. The mask for an individual priority *pri* is calculated by the macro LOG_MASK(*pri*); the mask for all priorities up to and including *toppri* is given by the macro LOG_UPTO(*toppri*). The default allows all priorities to be logged.

EXAMPLES

```
syslog(LOG_ALERT, "who: internal error 23");

openlog("ftpd", LOG_PID, LOG_DAEMON);
setlogmask(LOG_UPTO(LOG_ERR));
syslog(LOG_INFO, "Connection from host %d", CallingHost);

syslog(LOG_INFO|LOG_LOCAL2, "foobar error: %m");
```

SEE ALSO

logger(1), syslogd(8)

NAME

system – issue a shell command

SYNOPSIS

#include <stdlib.h>

int system(const char *string);

DESCRIPTION

system causes the *string* to be given to *sh*(1) as input as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

SEE ALSO

popen(3S), execve(2), wait(2)

DIAGNOSTICS

Exit status 127 indicates the shell couldn't be executed.

NAME

tpmount, tpunmount, tpqueue, tpattr, tplabel, tpunlabel, tpwait, tperror – tape subsystem calls

SYNOPSIS

```
#include <tape.h>

tpmount(mntopt)
struct tp_mntopt *mntopt;

tpunmount(symname, keeponline)
char *symname;
int keeponline;

struct queue_t *
tpqueue()

tpattr(symname, blocksize, recordsize, format, fileident, access, delim)
char *symname;
int blocksize, recordsize;
char format;
char *fileident;
int access;
int delim;

tplabel(symname, access)
char *symname;
char access;

tpunlabel(symname)
char *symname;

tplist(symname, labselect, flags)
char *symname;
int labselect;
int flags;

tpwait(symname)
char *symname;

tperror(s)
char *s;

char *tp_errlist[];
int tp_nerr;
```

DESCRIPTION

These functions make calls to the tape subsystem. Each of these routines perform the same function as the user commands (see *tpmount*(1), et al). These functions are obtained with the loader option **-ltape**.

tpmount() performs the same function as *tpmount*(1), except the request is always put into the background. *tpwait*() must be called to wait for the request to finish. The input structure is defined in *tape.h*:

```
struct tp_mntopt {
    int flags;          /* Mount flags (see TF_*)*/
    int skip;           /* Number of file to skip from beginning */
    char *label;        /* The label type (ansi, ibm, etc). */
    char *type;         /* Tape type */
    char *density;      /* Tape density */
    char *speed;        /* Tape speed */
    char *comment;      /* Comment for operator */
```

```
    char *sym_name;          /* Name for symbolic link to drive */
    char *vsns;              /* Volume serial numbers, separated by commas */
    char *dev_name;          /* Drive to allocate */
    struct tp_mntopt *next;  /* Pointer to the next tape in the list of */
                             /* tapes that need mounted simultaneously */
};

/* tpmount flags */
#define TF_MODE                  0007    /* the device mode: block, char, labeled */
#define   TM_DEFAULT             0000    /* use default device mode */
#define   TM_CHAR                0001    /* char special device */
#define   TM_BLOCK               0002    /* block special device */
#define   TM_LABEL               0003    /* labeled */
#define TF_READONLY              00010   /* tape should not have a write ring */
#define TF_READONLY_DEFAULT      00020   /* use default value */
#define TF_KEEPLOCAL             00040   /* keep tape mount on local host */
#define TF_KEEPLOCAL_DEFAULT     00100   /* use default value */
#define TF_REWINDDEV             00200   /* rewind device is desired */
#define TF_REWINDDEV_DEFAULT     00400   /* use default value */
#define TF_BYPASS                01000   /* bypass label processing */
#define TF_BYPASS_DEFAULT        02000   /* use default value */
#define TF_QUEUEALLOC            04000   /* queue the drive alloc */
```

tpqueue() returns the head of a linked list of requests that are queued in the tape subsystem. The file *tape.h* defines the *queue_t* structure:

```
/* list of queued & active tape requests */
struct queue_t {
    char *hostname;          /* host where drive is located */
    char *drive;             /* name of the drive including unit */
                             /* number */
    char *actualdevice;      /* real tape device name */
    char *userdevice;        /* user device name (eg, /dev/lt/u0) */
    int uid;                 /* user uid */
    char *vsn;               /* tape vsn */
    char *sym_name;          /* symbolic link name */
    struct queue_t *next;    /* pointer to next item in linked list */
};
```

tpunmount() performs the same function as *tpunmount*(1). If keeponline is non-zero, then the tape is not taken offline. This is equivalent to the -k option to *tpunmount*(1).

tpattr() performs the same function as *tpattr*(1).

The blocksize, recordsize, and format values are changed if their value is non-zero. Otherwise, the value is left unchanged. The fileident is changed if its value is not NULL and the string length is greater than zero. The access and delim values are changed if their value is not -1. The file *tape.h* defines several contants that can be used to set the access and delimiter values:

```
/* tpattr access bits */
#define TO_READ          04      /* others can read */
#define TO_WRITE         02      /* others can write */

/* tpattr delimiter options */
#define TD_BLOCKNL       0       /* block-newline delimiter */
#define TD_NEWLINE       1       /* newline delimiter */
```

```
#define TD_SYSCALL          2       /* system call delimiter */
```

tplabel() performs the same function as *tplabel*(1). The access argument can be either space or non-space. A space indicates that others are allowed to access this tape. Non-space indicates that others cannot access this tape.

tpunlabel() performs the same function as *tpunlabel*(1).

tplist() performs the same function as *tplist*(1). *tplist* returns a socket descriptor which must be read (until EOF) to get the label information. The format of the socket data is the same as the output of *tplist*(1). The file *tape.h* defines several contants that can be used to set the labselect and flags values:

```
/* tplist label select options */
#define TLS_DETAIL      001     /* detailed info on labels */
#define TLS_VOL         002     /* just volume info */
#define TLS_FILE        004     /* just file info */

/* tplist flag options */
#define TLF_RAW         001     /* display raw labels (with newlines) */
```

tpwait() performs the same function as *tpwait*(1).

tperror() prints a short message on the standard error file describing the last tape error.

RETURN VALUES

For *tpqueue*(), NULL is returned if the call fails. For all other calls, -1 is returned if the call fails. When a call fails, an error code if left in the global structure *tp_error*. *tperror*() can be used to interpret this error code.

SEE ALSO

tpmount(1), tpunmount(1), tpqueue(1), tpattr(1), tplabel(1), tpunlabel(1), tplist(1), tpwait(1)

NAME

tas – indivisibly test and set a memory location

SYNOPSIS

tas(addr)
caddr_t addr;

DESCRIPTION

tas indivisibly tests and sets the memory location *addr*. The return value represents the previous value of the location.

If the return value is zero, you have acquired the lock. If the return value is non-zero, sleep and try again.

SEE ALSO

mset(3), msleep(2)

NAME

tcgetattr, tcsetattr - get/set terminal characteristics

SYNOPSIS

#include <termios.h>

tcgetattr(fd, termios_p)
int fd;
struct termios *termios_p;

tcsetattr(fd, optional_action, termios_p)
int fd;
int optional_actions;
struct termios *termios_p;

DESCRIPTION

tcgetattr() gets the parameters associated with the object referred to by *fd* and stores them in the termios structure referenced by *termios_p*. This function is allowed from a background process; however, the terminal attributes may be subsequently changed by a foreground process.

The *tcsetattr()* function sets the parameters associated with the terminal from the termios structure referenced by *termios_p*.

optional_action determines when the change will occur as follows. If *optional_action* is TCSANOW, the change will occur immediately. If *optional_action* is TCSADRAIN, the change will occur after output written to *fd* has been transmitted to the terminal. This action should be used when changing parameters that affect output. If *optional_action* is TCSAFLUSH, the change will occur after all output has drained, and all input that has been received but not read will be discarded before the change.

TCSANOW, TCSADRAIN, and TCSAFLUSH are defined in **<termios.h>**.

RETURNS

Upon successful completion, a value of zero is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

If any of the following conditions occur, the function will return -1 and set *errno* to the corresponding value:

[EBADF] The *fd* argument is not a valid file descriptor.

[EINVAL] The *optional_action* argument is not a proper value.

[ENOTTY] The file associated with *fd* is not a terminal.

REFERENCES

cfgetispeed(3), cfsetispeed(3), cfgetospeed(3), cfsetospeed(3), tcgetpgrp(3), tcsetpgrp(3), termios(4).

NAME

tcgetpgrp, tcsetpgrp - get/set terminal process group

SYNOPSIS

#include <sys/types.h>

pid_t tcgetpgrp(fd)
int fd;

tcsetpgrp(fd, pgrp_id)
int fd;
pid_t pgrp_id;

DESCRIPTION

tcgetpgrp() returns the process group ID of the foreground process group associated with the terminal. The file associated with *fd* either must be the controlling terminal of the calling process or must be the master side of a pty device pair.

The *tcsetpgrp()* function sets the foreground process group ID associated with the terminal to *pgrp_id*. The file associated with *fd* must be the controlling terminal of the calling process, and the controlling terminal must be currently associated with the session of the calling process. The value of *pgrp_id* must match a process group ID of a process in the same session as the calling process.

RETURNS

tcgetpgrp() returns the process group ID of the foreground process group if successful; otherwise a value of –1 is returned and *errno* is set to indicate the error.

tcsetpgrp() returns a value of zero upon successful completion; otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

If any of the following conditions occur, the function will return –1 and set *errno* to the corresponding value:

[EBADF] The *fd* argument is not a valid file descriptor.

[EINVAL] The value of the *pgrp_id* argument is invalid.

[ENOTTY] The file associated with *fd* is neither the controlling terminal of the calling process nor the master side of a pty device pair (see *pty*(4)).

[EPERM] The value of the *pgrp_id* does not match the process group ID of a process in the same session as the calling process.

REFERENCES

intro(2), cfgetospeed(3), cfsetospeed(3), tcgetattr(3), tcsetattr(3), termios(4), pty(4), tty(4).

NAME

tcsendbreak, tcdrain, tcflush, tcflow - terminal line control functions

SYNOPSIS

```
#include <termios.h>

tcsendbreak(fd, duration)
int fd;
int duration;

tcdrain(fd)
int fd;

tcflush(fd, queue_selector)
int fd;
int queue_selector;

tcflow(fd, action)
int fd;
int action;
```

DESCRIPTION

If the terminal is using asynchronous serial data transmission, the *tcsendbreak()* function will transmit a break for a specific duration. If *duration* is zero, the break is transmitted for 0.25 seconds. If *duration* is non-zero it will be interpreted as the duration of the break in 0.1 second increments.

The *tcdrain()* function will wait until all output written to the object referred to by *fd* has been transmitted to the terminal.

The *tcflush()* function will discard data written to the object referred to by *fd* but not transmitted or data received but not read, depending on the value of *queue_selector*. If *queue_selector* is TCIFLUSH, data received but not read will be flushed. If *queue_selector* is TCOFLUSH, data written but not transmitted will be flushed. If *queue_selector* is TCIOFLUSH, both data received but not read and data written but not transmitted will be flushed.

The *tcflow()* function will suspend transmission or reception of data on the object referred to by *fd*, depending on the value of *action*. If *action* is TCOOFF, it will suspend output. If *action* is TCOON, it will restart suspended output. If *action* is TCIOFF, the system will transmit a STOP character, which is intended to cause the terminal device to stop transmitting data to the system. If *action* is TCION, the system will transmit a START character, which is intended to cause the terminal device to start transmitting data to the system.

Values of *queue_selector* and *action* are defined in **<termios.h>**.

RETURNS

Upon successful completion, a value of zero is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

If any of the following conditions occur, the function will return -1 and set *errno* to the corresponding value:

[EBADF]	The *fd* argument is not a valid file descriptor.
[EINTR]	The signal interrupted the *tcdrain()* function.
[EINVAL]	The *queue_selector* or *action* argument is not a proper value.
[ENOTTY]	The file associated with *fd* is not a terminal.

REFERENCES

tcsetattr(3).

NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs – terminal independent operation routines

SYNOPSIS

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, **area;

char *
tgoto(cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

DESCRIPTION

These functions extract and use capabilities from the terminal capability data base *termcap*(5). These are low level routines; see *curses*(3X) for a higher level package.

Tgetent extracts the entry for terminal *name* into the buffer at *bp*. *Bp* should be a character buffer of size 1024 and must be retained through all subsequent calls to *tgetnum*, *tgetflag*, and *tgetstr*. *Tgetent* returns –1 if it cannot open the *termcap* file, 0 if the terminal name given does not have an entry, and 1 if all goes well. It will look in the environment for a TERMCAP variable. If found, and the value does not begin with a slash, and the terminal type **name** is the same as the environment string TERM, the TERMCAP string is used instead of reading the termcap file. If it does begin with a slash, the string is used as a path name rather than */etc/termcap*. This can speed up entry into programs that call *tgetent*, as well as to help debug new terminal descriptions or to make one for your terminal if you can't write the file */etc/termcap*.

Tgetnum gets the numeric value of capability *id*, returning –1 if is not given for the terminal. *Tgetflag* returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. *Tgetstr* gets the string value of capability *id*, placing it in the buffer at *area*, advancing the *area* pointer. It decodes the abbreviations for this field described in *termcap*(5), except for cursor addressing and padding information.

Tgoto returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables **UP** (from the **up** capability) and **BC** (if **bc** is given rather than **bs**) if necessary to avoid placing **\n**, **^D** or **^@** in the returned string. (Programs which call tgoto should be sure to turn off the XTABS bit(s), since *tgoto* may now output a tab. Note that programs using termcap should in general turn off XTABS anyway since some terminals use control I for other functions, such as nondestructive space.) If a **%** sequence is given which is not understood, then *tgoto* returns "OOPS".

Tputs decodes the leading padding information of the string *cp*; *affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable, *outc* is a routine which is called with each character in turn. The external variable *ospeed* should contain the output speed of the terminal as encoded by *stty*(3). The external variable **PC** should contain a pad character to be used (from the **pc** capability) if a null (^@) is inappropriate.

FILES

/usr/lib/libtermcap.a	–ltermcap library
/etc/termcap	data base

SEE ALSO

ex(1), curses(3X), termcap(5)

AUTHOR

William Joy

NAME

time - get system time

SYNOPSIS

#include <time.h>

time_t time(time_t *tloc);

DESCRIPTION

The *time()* function returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds. If *tloc* is nonnull, the return value is also stored in the place to which *tloc* points.

RETURN VALUE

The *time()* function always succeeds and returns the value described above.

SEE ALSO

date(1), gettimeofday(2), ctime(3)

NAME

times – get process times

SYNOPSIS

#include <sys/times.h>

clock_t times(buffer)
struct tms *buffer;

DESCRIPTION

times() returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in **CLK_TCK**ths of a second.

The members of *struct tms* are:

tms_utime	User CPU time.
tms_ctime	System CPU time.
tms_cutime	User CPU time of terminated child processes.
tms_cstime	System CPU time of terminated child processes.

Note that the children times are the sum of the children's process times and their children's times.

RETURN VALUE

Upon success, the elapsed real time in **CLK_TCK**ths of a second, since system startup time is returned. This point does not change from one invocation of *times()* within the process to another. The return value may overflow the possible range of type *clock_t*. If the times function fails, a value of ((*clock_t*)–1) is returned, and *errno* is set to indicate the error.

ERRORS

It is unlikely that *times()* will fail. Any possible failures come from *getrusage(2)*.

BACKWARD COMPATIBILITY

Previous versions of *times()* filled the *struct tms* members with units measured in 60ths of a second. **CLK_TCK**ths are not necessarily the same.

Previous versions of *times()* always returned zero or –1.

SEE ALSO

time(1), getrusage(2), wait(2), time(3)

NAME

tmpfile, tmpnam – create a temporary file or generate a unique file name.

SYNOPSIS

```
#include <stdio.h>

FILE *tmpfile(void);

char *tmpnam(char *buffer);
```

DESCRIPTION

tmpfile creates a temporary file on */tmp*. This file is opened in mode "w+". The file is removed when it is closed; this occurs explicitly by calling *fclose* or implicitly by calling *exit*. If the program terminates abnormally, the file may not be removed.

tmpnam generates a string that is a unique file name. If *buffer* is not NULL, the string will be stored in *buffer;* otherwise it will be stored in a static buffer internal to *tmpnam*. Subsequent calls to *tmpnam* will overwrite this static buffer. If *buffer* is not NULL, it must identify an array of at least L_tmpnam characters. Each of the first TMP_MAX calls to *tmpnam* will generate different strings.

RETURNS

tmpfile returns a pointer to the FILE structure for the open file. If the file cannot be opened, a null pointer is returned.

tmpnam returns a pointer to the string.

SEE ALSO

fopen(3s).

NAME

ttyname, isatty – find name of a terminal

SYNOPSIS

char *ttyname(filedes)
int filedes;

isatty(filedes)
int filedes;

DESCRIPTION

ttyname returns a pointer to the null-terminated path name of the terminal device associated with file descriptor *filedes* (this is a system file descriptor and has nothing to do with the standard I/O FILE typedef).

isatty returns 1 if *filedes* is associated with a terminal device, 0 otherwise.

FILES

/dev/*
/etc/ttys

SEE ALSO

ioctl(2), ttys(5)

DIAGNOSTICS

ttyname returns a null pointer (0) if *filedes* does not describe a terminal device in directory */dev*.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

tzset – set time zone information

SYNOPSIS

#include <time.h>

void tzset()

DESCRIPTION

The *tzset()* function uses the value of the environment variable **TZ** to set time conversion information used by *localtime(), ctime(), strftime(),* and *mktime().* If **TZ** is absent from the environment, implementation-defined default time zone information is used.

The *tzset()* function sets the external variable *tzname:*

extern char *tzname[2] = { "*std*", "*dst*" };

where *std* and *dst* are the standard and summer time zone names defined form the environment variable **TZ.**

The value of **TZ** has the following form (spaces have been inserted for clarity):

std offset dst offset, rule

Where:

std and *dst* are three or more byte sequences that are the designation for the standard (*std*) or summer (*dst*) time zone. Only *std* is required; if *dst* is missing, then summer time does not apply in this locale. Upper- and lowercase letters are explicitly allowed. Any characters except a leading colon (:), digits, comma(,), minus(-), plus(+), and ASCII NULL are allowed.

offset indicates the value one must add to the local time to arrive at Coordinated Universal Time. The *offset* has the form:

hh[:mm[:ss]]

The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required and may be a single digit. The *offset* following *std* is required. If no *offset* follows *dst,* summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour shall be between zero and 24, and the minutes (and seconds) - if present - between zero and 59. Out of range values will cause unpredictable behavior. If preceded by a '-', the time zone is east of the Prime Meridian; otherwise it is west (which may be indicated by an optional preceding '+').

Rule indicated when to change to and back from summer time. The *rule* has the form:

date/time,date/time

where the first *date* describes when the change from standard to summer occurs, and the second *date* describes when the change back happens. Each *time* field describes when, in current local time, the change to the other time is made. The format of *date* shall be one of the following:

J*n*, the Julian day *n* (one based). Because leap days are not counted, it is impossible to refer to February 29.

n, The zero-based Julian day. Because leap days are counted, it is possible to refer to February 29.

M*m.n.d*, day *d* of week *n* of month *m*. Week *n* is the *n*th week in which day *d* appears. Week 5 is always the last week. Sunday is day zero.

The *time* has the same format as *offset* except that no leading sign ('-' or '+') is allowed. The default, if *time* is not given, is 02:00:00.

NAME

ungetc – push character back into input stream

SYNOPSIS

#include <stdio.h>

int ungetc(int c, FILE *stream)

DESCRIPTION

ungetc pushes the character *c* back into an input stream. That character will be returned by the next *getc* call on that stream. *ungetc* returns *c*.

One character of pushback is guaranteed from the stream. Attempts to push EOF are rejected.

fseek(3S) erases all memory of pushed back characters.

SEE ALSO

getc(3S), setbuf(3S), fseek(3S)

DIAGNOSTICS

Ungetc returns **EOF** if it can't push a character back.

NAME

utime – set file times

SYNOPSIS

```
#include <sys/types.h>
#include <utime.h>

int utime(file, timep)
char *file;
struct utimbuf *timep;
```

DESCRIPTION

The *utime()* function uses the *actime* and *modtime* members of the *utimbuf* structure pointed to by *timep* to set the recorded times for *file*. If *timep* is NULL, the current time is used.

The caller's effective uid must match the owner of the file, or the caller must be the super-user. The "inode-changed" time of the file is set to the current time.

RETURNS

On success, a value of zero is returned. Otherwise, –1 is returned and *errno* is set to indicate the error.

ERRORS

utime() fails if:

[EACCES]	The effective user ID is not super-user and not the owner of the file and *times* is NULL and write access is denied.
[EACCES]	A component of the path prefix denies search permission.
[EFAULT]	The *file* argument points outside the process's allocated address space.
[EINVAL]	The pathname contained a character with the high-order bit set.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[ENAMETOOLONG]	The pathname was too long.
[ENIENT]	The pathname argument pointed to an empty string.
[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The process is not super-user and not the owner of the file.
[EROFS]	The file system containing the file is mounted read-only.

SEE ALSO

utimes(2), stat(2)

NAME

valloc – aligned memory allocator

SYNOPSIS

char *valloc(size)
unsigned size;

DESCRIPTION

Valloc allocates *size* bytes aligned on a page boundary. It is implemented by calling *malloc*(3) with a slightly larger request, saving the true beginning of the block allocated, and returning a properly aligned pointer.

DIAGNOSTICS

Valloc returns a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block.

BUGS

Vfree isn't implemented.

NAME

varargs – variable argument list

SYNOPSIS

```
#include <varargs.h>

function(va_alist)
va_dcl
va_list pvar;
va_start(pvar);
f = va_arg(pvar, type);
va_end(pvar);
```

DESCRIPTION

This set of macros provides a means of writing portable procedures that accept variable argument lists. Routines having variable argument lists (such as *printf*(3)) that do not use varargs are inherently nonportable, since different machines use different argument passing conventions.

va_alist is used in a function header to declare a variable argument list.

va_dcl is a declaration for **va_alist**. Note that there is no semicolon after **va_dcl.**

va_list is a type which can be used for the variable *pvar*, which is used to traverse the list. One such variable must always be declared.

va_start(pvar) is called to initialize *pvar* to the beginning of the list.

va_arg(*pvar, type*) will return the next argument in the list pointed to by *pvar*. *Type* is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, since it cannot be determined at runtime.

va_end(*pvar*) is used to finish up.

Multiple traversals, each bracketed by **va_start** ... **va_end,** are possible.

EXAMPLE

```
#include <varargs.h>
execl(va_alist)
va_dcl
{
        va_list ap;
        char *file;
        char *args[100];
        int argno = 0;

        va_start(ap);
        file = va_arg(ap, char *);
        while (args[argno++] = va_arg(ap, char *))
                ;
        va_end(ap);
        return execv(file, args);
}
```

BUGS

It is up to the calling routine to determine how many arguments there are, since it is not possible to determine this from the stack frame. For example, *execl* passes a 0 to signal the end of the list. *Printf* can tell how many arguments are supposed to be there by the format.

NAME

vlimit – control maximum system resource consumption

SYNOPSIS

#include <sys/vlimit.h>

vlimit(resource, value)

DESCRIPTION

This facility is superseded by getrlimit(2).

Limits the consumption by the current process and each process it creates to not individually exceed *value* on the specified *resource*. If *value* is specified as –1, then the current limit is returned and the limit is unchanged. The resources which are currently controllable are:

LIM_NORAISE	A pseudo-limit; if set non-zero then the limits may not be raised. Only the super-user may remove the *noraise* restriction.
LIM_CPU	the maximum number of cpu-seconds to be used by each process
LIM_FSIZE	the largest single file which can be created
LIM_DATA	the maximum growth of the data+stack region via *sbrk*(2) beyond the end of the program text
LIM_STACK	the maximum size of the automatically-extended stack region
LIM_CORE	the size of the largest core dump that will be created.
LIM_MAXRSS	a soft limit for the amount of physical memory (in bytes) to be given to the program. If memory is tight, the system will prefer to take memory from processes which are exceeding their declared LIM_MAXRSS.

Because this information is stored in the per-process information this system call must be executed directly by the shell if it is to affect all future processes created by the shell; *limit* is thus a built-in command to *csh*(1).

The system refuses to extend the data or stack space when the limits would be exceeded in the normal way; a *break* call fails if the data space limit is reached, or the process is killed when the stack limit is reached (since the stack cannot be extended, there is no way to send a signal!).

A file i/o operation which would create a file which is too large will cause a signal SIGXFSZ to be generated, this normally terminates the process, but may be caught. When the cpu time limit is exceeded, a signal SIGXCPU is sent to the offending process; to allow it time to process the signal it is given 5 seconds grace by raising the cpu time limit.

SEE ALSO

csh(1)

BUGS

If LIM_NORAISE is set, then no grace should be given when the cpu time limit is exceeded.

There should be *limit* and *unlimit* commands in *sh*(1) as well as in *csh*.

This call is peculiar to this version of UNIX. The options and specifications of this system call and even the call itself are subject to change. It may be extended or replaced by other facilities in future versions of the system.

NAME

vprintf, vfprintf, vsprintf – stdarg style formatted output conversion

SYNOPSIS

```
#include <stdio.h>

#include <stdarg.h>

int vprintf(const char *format, va_list arg );

int vfprintf(FILE *stream, const char *format, va_list arg );

int vsprintf(char *s, const char *format, va_list arg );
```

DESCRIPTION

vprintf, vfprintf, and *vsprintf* are alternate forms of *printf, fprintf,* and *sprintf. vprintf, vfprintf,* and *vsprintf* require a single argument of type *va_list* rather than a variable number of arguments. This argument must be initialized with the *va_start* macro provided in stdarg.h.

BACKWARD COMPATIBILITY

When compiling or linking with the *-pcc* option of the **cc** command, different library routines are linked. The routines described above differ from these backward-compatible libraries in the following way:

- None of these routines are available in the backward-compatible mode.

SEE ALSO

stdarg.h(3) printf(3s)

NAME

vtimes – get information about resource utilization

SYNOPSIS

vtimes(par_vm, ch_vm)
struct vtimes *par_vm, *ch_vm;

DESCRIPTION

This facility is superseded by getrusage(2).

Vtimes returns accounting information for the current process and for the terminated child processes of the current process. Either *par_vm* or *ch_vm* or both may be 0, in which case only the information for the pointers which are non-zero is returned.

After the call, each buffer contains information as defined by the contents of the include file */usr/include/sys/vtimes.h:*

```
struct vtimes {
        int     vm_utime;               /* user time (*HZ) */
        int     vm_stime;               /* system time (*HZ) */
        /* divide next two by utime+stime to get averages */
        unsigned vm_idsrss;             /* integral of d+s rss */
        unsigned vm_ixrss;              /* integral of text rss */
        int     vm_maxrss;              /* maximum rss */
        int     vm_majflt;              /* major page faults */
        int     vm_minflt;              /* minor page faults */
        int     vm_nswap;               /* number of swaps */
        int     vm_inblk;               /* block reads */
        int     vm_oublk;               /* block writes */
};
```

The *vm_utime* and *vm_stime* fields give the user and system time respectively in 60ths of a second (or 50ths if that is the frequency of wall current in your locality.) The *vm_idrss* and *vm_ixrss* measure memory usage. They are computed by integrating the number of memory pages in use each over cpu time. They are reported as though computed discretely, adding the current memory usage (in 4096 byte pages) each time the clock ticks. If a process used 5 core pages over 1 cpu-second for its data and stack, then *vm_idsrss* would have the value 5*60, where *vm_utime+vm_stime* would be the 60. *Vm_idsrss* integrates data and stack segment usage, while *vm_ixrss* integrates text segment usage. *Vm_maxrss* reports the maximum instantaneous sum of the text+data+stack core-resident page count.

The *vm_majflt* field gives the number of page faults which resulted in disk activity; the *vm_minflt* field gives the number of page faults incurred in simulation of reference bits; *vm_nswap* is the number of swaps which occurred. The number of file system input/output events are reported in *vm_inblk* and *vm_oublk* These numbers account only for real i/o; data supplied by the caching mechanism is charged only to the first process to read or write the data.

SEE ALSO

time(2), wait(2)

BUGS

This call is peculiar to the Berkeley version of UNIX. The options and specifications of this system call are subject to change. It may be extended to include additional information in future versions of the system.