Part Number 800-1108-01
Revision: E of 7th January 1984
For: Sun System Release 1.1

System Interface Manual

for the

Sun Workstation

Sun Microsystems, Inc.,
2550 Garcia Avenue
Mountain View
California 94043
(415) 960-1300

Credits and Acknowledgements

This manual is composed of parts of the original UNIX Programmer's Manual, plus two other papers from University of California at Berkeley. The authors' names and the titles of the original works appear here.

Interprocess Communication Primer
: is based on the document *4.2bsd Interprocess Communications Primer* by Samuel J. Leffler, Robert S. Fabry and William N. Joy, of the Computer Systems Research Group, U.C. Berkeley.

System Interface Overview
: is based on the *4.2BSD System Interface Overview* by William Joy, Eric Cooper, Robert Fabry, Samuel Leffler, Kirk McKusick and David Mosher; released by the Computer Systems Research Group at U.C. Berkeley in July, 1983.

Trademarks

Revision History

Revision	Date	Comments
A	23 February 1983	First release of this Manual.
B	15 April 1983	Second Release of this manual involved many corrections to manual pages.
C	1 August 1983	Third Release of this manual involved many corrections to manual pages. Added glossary of system calls and system error responses.
D	1 November 1983	Fourth Release of this manual involved many corrections to manual pages. Fixed numerous incorrect cross-references between pages. Added a *System Interface Overview* and the *Interprocess Communication Primer* as a tutorial.
E	7 January 1984	Fifth Release of this manual involved many corrections to manual pages.

System Interface Manual

Table of Contents

PERMUTED INDEX

System Interface Overview

Table of Contents

System Interface Overview

Revised for Sun Release 1.1, April 1984

This document summarizes the facilities provided by the 1.1 and later releases of the UNIX† operating system for the Sun Workstation. It does not attempt to act as a tutorial for use of the system nor does it attempt to explain or justify the design of the system facilities. It gives neither motivation nor implementation details, in favor of brevity. This document is in three major parts:

Part I describes the basic kernel functions provided to a UNIX process: process naming and protection, memory management, software interrupts, object references (descriptors), time and statistics functions, and resource controls. These facilities, as well as facilities for bootstrap, shutdown and process accounting, are provided solely by the kernel.

Part II describes the standard system abstractions for files and file systems, communication, terminal handling, and process control and debugging. These facilities are implemented by the operating system or by network server processes.

Part III is an appendix containing a summary of the facilities described in parts I and II.

Notation and Types

The notation used to describe system calls is a variant of a C language call, consisting of a prototype call followed by declaration of parameters and results. An additional keyword **result**, not part of the normal C language, is used to indicate which of the declared entities receive results. As an example, consider the *read* call, as described in section 8.1:

```
cc = read(fd, buf, nbytes);
result int cc; int fd; result char *buf; int nbytes;
```

The first line shows how the *read* routine is called, with three parameters. As shown on the second line *cc* is an integer and *read* also returns information in the parameter *buf*.

Description of all error conditions arising from each system call is not provided here; they appear in the *System Interface Manual*. In particular, when accessed from the C language, many calls return a characteristic -1 value when an error occurs, returning the error code in the global variable *errno*. Other languages may present errors in different ways.

A number of system standard types are defined in the *<sys/types.h>* include file and used in the specifications here and in many C programs. These include **caddr_t** giving a memory

† UNIX is a trademark of Bell Laboratories.

address (typically as a character pointer), **off_t** giving a file offset (typically as a long integer), and a set of unsigned types **u_char**, **u_short**, **u_int** and **u_long**, shorthand names for **unsigned char**, **unsigned short**, etc.

Part I — Kernel Primitives

The facilities available to a UNIX user process are logically divided into two parts: kernel facilities directly implemented by UNIX code running in the operating system, and system facilities implemented either by the system, or in cooperation with a *server process*. The kernel facilities are described in this part of the document.

The facilities implemented in the kernel are those which define the *UNIX virtual machine* which each process runs in. Like many real machines, this virtual machine has memory management hardware, an interrupt facility, timers and counters. The UNIX virtual machine also allows access to files and other objects through a set of *descriptors*. Each descriptor resembles a device controller, and supports a set of operations. Like devices on real machines, some of which are internal to the machine and some of which are external, parts of the descriptor machinery are built-in to the operating system, while other parts are often implemented in server processes on other machines. The facilities provided through the descriptor machinery are described in Part II.

1. Processes and Protection

1.1. Host and Process Identifiers

Each UNIX host has associated with it a 32-bit host id, and a host name of up to 255 characters. These are set (by a privileged user) and returned by the calls:

```
sethostid(hostid);
long hostid;

hostid = gethostid();
result long hostid;

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

gethostname(buf, buflen);
result char *buf; int buflen;
```

The host id is not used in this release of the system. The *buf* containing the host name returned by *gethostname* is null-terminated (if space allows).

On each host runs a set of *processes*. Each process is largely independent of other processes, having its own protection domain, address space, timers, and an independent set of references to system or user implemented objects.

Each process in a host is named by an integer called the *process id*. This number is in the range 1-30000 and is returned by the *getpid* routine:

```
pid = getpid();
result int pid;
```

On each UNIX host this identifier is guaranteed to be unique; in a multi-host environment, the (hostid, process id) pairs are guaranteed unique.

1.2. Process Creation and Termination

A new process is created by making a logical duplicate of an existing process:

```
pid = fork();
result int pid;
```

The *fork* call returns twice, once in the parent process, where *pid* is the process identifier of the child, and once in the child process where *pid* is 0. The parent-child relationship induces a hierarchical structure on the set of processes in the system.

A process may terminate by executing an *exit* call:

```
exit(status);
int status;
```

returning 8 bits of exit status to its parent.

When a child process exits or terminates abnormally, the parent process receives information about any event which caused termination of the child process. A second call provides a non-blocking interface and may also be used to retrieve information about resources consumed by the process during its lifetime.

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

pid = wait(astatus);
result int pid; result union wait *astatus;

pid = wait3(astatus, options, arusage);
result int pid; result union waitstatus *astatus;
int options; result struct rusage *arusage;
```

A process can overlay itself with the memory image of another process, passing the newly created process a set of parameters, using the call:

```
execve(name, argv, envp)
char *name, **argv, **envp;
```

The specified *name* must be a file which is in a format recognized by the system, either a binary executable file or a file which causes the execution of a specified interpreter program to process its contents.

1.3. User and Group Ids

Each process in the system has associated with it two user-id's: a *real user id* and a *effective user id*, both non-negative 16 bit integers. Each process has an *real accounting group id* and an *effective accounting group id* and a set of *access group id's.* The group id's are non-negative 16 bit integers. Each process may be in several different access groups, with the maximum concurrent number of access groups a system compilation parameter, the constant NGROUPS in the file <sys/param.h>, guaranteed to be at least 8.

The real and effective user ids associated with a process are returned by:

```
ruid = getuid();
result int ruid;

euid = geteuid();
result int euid;
```

the real and effective accounting group ids by:

```
rgid = getgid();
result int rgid;

egid = getegid();
result int egid;
```

and the access group id set is returned by a *getgroups* call:

```
ngroups = getgroups(gidsetsize, gidset);
result int ngroups; int gidsetsize; result int gidset[gidsetsize];
```

The user and group id's are assigned at login time using the *setreuid*, *setregid*, and *setgroups* calls:

```
setreuid(ruid, euid);
int ruid, euid;

setregid(rgid, egid);
int rgid, egid;

setgroups(gidsetsize, gidset);
int gidsetsize; int gidset[gidsetsize];
```

The *setreuid* call sets both the real and effective user-id's, while the *setregid* call sets both the real and effective accounting group id's. Unless the caller is the super-user, *ruid* must be equal to either the current real or effective user-id, and *rgid* equal to either the current real or effective accounting group id. The *setgroups* call is restricted to the super-user.

1.4. Process Groups and System Terminals

Each process in the system is also normally associated with a *process group*. The group of processes in a process group is sometimes referred to as a *job* and manipulated by high-level system software (such as the shell). The current process group of a process is returned by the *getpgrp* call:

```
pgrp = getpgrp(pid);
result int pgrp; int pid;
```

The process group associated with a process may be changed by the *setpgrp* call:

```
setpgrp(pid, pgrp);
int pid, pgrp;
```

Newly created processes are assigned process id's distinct from all processes and process groups, and the same process group as their parent. A normal (unprivileged) process may set its process group equal to its process id. A privileged process may set the process group of any process to any value.

When a process is in a specific process group it may receive software interrupts affecting the group, causing the group to suspend or resume execution or to be interrupted or terminated. In particular, every system terminal has a process group and only processes which are in the process group of a terminal may read from the terminal, allowing arbitration of terminals among several different jobs. A process can examine the process group of a terminal via the *ioctl* call:

```
ioctl(fd, TIOCGPGRP, pgrp);
int fd; result int *pgrp;
```

A process may change the process group of any terminal which it can write by the *ioctl* call:

```
ioctl(fd, TIOCSPGRP, pgrp);
int fd; int *pgrp;
```

The terminal's process group may be set to any value. Thus, more than one terminal may be in a process group.

Each process in the system is usually associated with a *control terminal*, accessible through the file **/dev/tty**. A newly created process inherits the control terminal of its parent. A process may be in a different process group than its control terminal, in which case the process does not receive software interrupts affecting the control terminal's process group.

2. Memory management

This section represents the interface planned for later releases of the system. Of the calls described in this section, only *sbrk*, *etpagesize*, and *mmap* are included in the current release. Note that *mmap* is restricted in that it only works with certain character devices such as the framebuffer and devices like *mbmem*.

2.1. Text, Data, and Stack

Each process begins execution with three logical areas of memory called text, data and stack. The text area is read-only and shared, while the data and stack areas are private to the process. Both the data and stack areas may be extended and contracted on program request. The call

```
addr = sbrk(incr);
result caddr_t addr; int incr;
```

changes the size of the data area by *incr* bytes and returns the new end of the data area, while

```
addr = sstk(incr);
result caddr_t addr; int incr;
```

changes the size of the stack area. The stack area is also automatically extended as needed. On the VAX the text and data areas are adjacent in the P0 region, while the stack section is in the P1 region, and grows downward.

2.2. Mapping Pages

The system supports sharing of data between processes by allowing pages to be mapped into memory. These mapped pages may be *shared* with other processes or *private* to the process. Protection and sharing options are defined in <mman.h> as:

```
/* protections are chosen from these bits, or-ed together */
#define PROT_READ     0x4   /* pages can be read */
#define PROT_WRITE    0x2   /* pages can be written */
#define PROT_EXEC     0x1   /* pages can be executed */

/* sharing types; choose either SHARED or PRIVATE */
#define MAP_SHARED    1     /* share changes */
#define MAP_PRIVATE   2     /* changes are private */
```

The cpu-dependent size of a page is returned by the *getpagesize* system call:

```
pagesize = getpagesize();
result int pagesize;
```

The call:

```
mmap(addr, len, prot, share, fd, pos);
caddr_t addr; int len, prot, share, fd; off_t pos;
```

causes the pages starting at *addr* and continuing for *len* bytes to be mapped from the object represented by descriptor *fd*, at absolute position *pos*. The parameter *share* specifies whether modifications made to this mapped copy of the page, are to be kept *private*, or are to be *shared* with other references. The parameter *prot* specifies the accessibility of the mapped pages. The *addr*, *len*, and *pos* parameters must all be multiples of the pagesize.

A process can move pages within its own memory by using the *mremap* call:

```
mremap(addr, len, prot, share, fromaddr);
caddr_t addr; int len, prot, share; caddr_t fromaddr;
```

This call maps the pages starting at *fromaddr* to the address specified by *addr*.

A mapping can be removed by the call

```
munmap(addr, len);
caddr_t addr; int len;
```

This causes further references to these pages to refer to private pages initialized to zero.

2.3. Page Protection Control

A process can control the protection of pages using the call

```
mprotect(addr, len, prot);
caddr_t addr; int len, prot;
```

This call changes the specified pages to have protection *prot*.

2.4. Giving and Getting Advice

A process that has knowledge of its memory behavior may use the *madvise* call:

```
madvise(addr, len, behav);
caddr_t addr; int len, behav;
```

Behav describes expected behavior, as given in <mman.h>:

```
#define MADV_NORMAL     0      /* no further special treatment */
#define MADV_RANDOM     1      /* expect random page references */
#define MADV_SEQUENTIAL        2/* expect sequential references */
#define MADV_WILLNEED 3        /* will need these pages */
#define MADV_DONTNEED 4        /* don't need these pages */
```

Finally, a process may obtain information about whether pages are core resident by using the call

```
mincore(addr, len, vec);
caddr_t addr; int len; result char *vec;
```

Here the current core residency of the pages is returned in the character array *vec*, with a value of 1 meaning that the page is in-core.

3. Signals

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 the *handler* to which a signal is delivered, or specify that the signal is to be *blocked* or *ignored*. A process may also specify that a *default* action is to be taken when signals occur.

Some signals will cause a process to exit when they are not caught. This may be accompanied by creation of a *core* image file, containing the current memory image of the process for use in post-mortem debugging. A process may choose to have signals delivered on a special stack, so that sophisticated software stack manipulations are possible.

All signals have the same *priority*. If multiple signals are pending simultaneously, the order in which they are delivered to a process is implementation specific. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. Mechanisms are provided whereby critical sections of code may protect themselves against the occurrence of specified signals.

3.1. Signal Types

The signals defined by the system fall into one of five classes: hardware conditions, software conditions, input/output notification, process control, or resource control. The set of signals is defined in the file <signal.h>.

Hardware signals are derived from exceptional conditions which may occur during execution. Such signals include SIGFPE representing floating point and other arithmetic exceptions, SIGILL for illegal instruction execution, SIGSEGV for addresses outside the currently assigned area of memory, and SIGBUS for accesses that violate memory protection constraints. Other, more cpu-specific hardware signals exist, such as those for the various customer-reserved instructions on the VAX (SIGIOT, SIGEMT, and SIGTRAP).

Software signals reflect interrupts generated by user request: SIGINT for the normal interrupt signal; SIGQUIT for the more powerful *quit* signal, that normally causes a core image to be generated; SIGHUP and SIGTERM that cause graceful process termination, either because a user has "hung up", or by user or program request; and SIGKILL, a more powerful termination signal which a process cannot catch or ignore. Other software signals (SIGALRM, SIGVTALRM, SIGPROF) indicate the expiration of interval timers.

A process can request notification via a SIGIO signal when input or output is possible on a descriptor, or when a *non-blocking* operation completes. A process may request to receive a SIGURG signal when an urgent condition arises.

A process may be *stopped* by a signal sent to it or the members of its process group. The SIGSTOP signal is a powerful stop signal, because it cannot be caught. Other stop signals SIGTSTP, SIGTTIN, and SIGTTOU are used when a user request, input request, or output request respectively is the reason the process is being stopped. A SIGCONT signal is sent to a process when it is continued from a stopped state. Processes may receive notification with a SIGCHLD signal when a child process changes state, either by stopping or by terminating.

Exceeding resource limits may cause signals to be generated. SIGXCPU occurs when a process nears its CPU time limit and SIGXFSZ warns that the limit on file size creation has been reached.

3.2. Signal Handlers

A process has a handler associated with each signal that controls the way the signal is delivered. The call

```
#include <signal.h>

struct sigvec {
        int         (*sv_handler)();
        int         sv_mask;
        int         sv_onstack;
};

sigvec(signo, sv, osv)
int signo; struct sigvec *sv; result struct sigvec *osv;
```

assigns interrupt handler address *sv_handler* to signal *signo*. Each handler address specifies either an interrupt routine for the signal, that the signal is to be ignored, or that a default action (usually process termination) is to occur if the signal occurs. The constants SIG_IGN and SIG_DEF used as values for *sv_handler* cause ignoring or defaulting of a condition. The *sv_mask* and *sv_onstack* values specify the signal mask to be used when the handler is invoked and whether the handler should operate on the normal run-time stack or a special signal stack (see below). If *osv* is non-zero, the previous signal vector is returned.

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 will be delivered. The process of signal delivery adds the signal to be delivered and those signals specified in the associated signal handler's *sv_mask* to a set of those *masked* for the process, saves the current process context, and places the process in the context of the signal handling routine. The call is arranged so that if the signal handling routine exits normally the signal mask will be restored and the process will resume execution in the original context. If the process wishes to resume in a different context, then it must arrange to restore the signal mask itself.

The mask of *blocked* signals is independent of handlers for signals. It prevents signals from being delivered much as a raised hardware interrupt priority level prevents hardware interrupts. Preventing an interrupt from occurring by changing the handler is analogous to disabling a device from further interrupts.

The signal handling routine *sv_handler* is called by a C call of the form

```
(*sv_handler)(signo, code, scp);
int signo; long code; struct sigcontext *scp;
```

The *signo* gives the number of the signal that occurred, and the *code*, a word of information supplied by the hardware. The *scp* parameter is a pointer to a machine-dependent structure containing the information for restoring the context before the signal.

3.3. Sending Signals

A process can send a signal to another process or group of processes with the calls:

```
kill(pid, signo);
int pid, signo;

killpgrp(pgrp, signo);
int pgrp, signo;
```

Unless the process sending the signal is privileged, it and the process receiving the signal must have the same effective user id.

Signals are also sent implicitly from a terminal device to the process group associated with the terminal when certain input characters are typed.

3.4. Protecting Critical Sections

To block a section of code against one or more signals, a *sigblock* call may be used to add a set of signals to the existing mask, returning the old mask:

```
oldmask = sigblock(mask);
result long oldmask; long mask;
```

The old mask can then be restored later with *sigsetmask*,

```
oldmask = sigsetmask(mask);
result long oldmask; long mask;
```

The *sigblock* call can be used to read the current mask by specifying an empty *mask*.

It is possible to check conditions with some signals blocked, and then to pause waiting for a signal and restoring the mask, by using:

```
sigpause(mask);
long mask;
```

3.5. Signal Stacks

Applications that maintain complex or fixed size stacks can use the call

```
struct sigstack {
        caddr_t     ss_sp;
        int         ss_onstack;
};

sigstack(ss, oss)
struct sigstack *ss; result struct sigstack *oss;
```

to provide the system with a stack based at *ss_sp* for delivery of signals. The value *ss_onstack* indicates whether the process is currently on the signal stack, a notion maintained in software by the system.

When a signal is to be delivered, the system checks whether the process is on a signal stack. If not, then the process is switched to the signal stack for delivery, with the return from the signal arranged to restore the previous stack.

If the process wishes to take a non-local exit from the signal routine, or run code from the signal stack that uses a different stack, a *sigstack* call should be used to reset the signal stack.

4. Timers

4.1. Real Time

The system's notion of the current Greenwich time and the current time zone is set and returned by the calls:

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

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

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

where the structures are defined in *<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 */
};
```

Earlier versions of UNIX contained only a 1-second resolution version of this call, which remains as a library routine:

```
time(tvp)
result long *tvp;
```

or

```
tv = time(0);
result long tv;
```

returning only the tv_sec field from the *gettimeofday* call.

4.2. Interval Time

The system provides each process with three interval timers, defined in <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 */
```

The ITIMER_REAL timer decrements in real time. It could be used by a library routine to maintain a wakeup service queue. 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 processes to statistically profile their execution. A SIGPROF signal is delivered when it expires.

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

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

and a timer is set or read by the call:

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

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

The third argument to *setitimer* specifies an optional structure to receive the previous contents of the interval timer. A timer can be disabled by specifying a timer value of 0.

The system rounds argument timer intervals to be not less than the resolution of its clock. This clock resolution can be determined by loading a very small value into a timer and reading the timer back to see what value resulted.

The *alarm* system call of earlier versions of UNIX is provided as a library routine using the ITIMER_REAL timer. The process profiling facilities of earlier versions of UNIX remain because it is not always possible to guarantee the automatic restart of system calls after receipt of a signal.

```
profil(buf, bufsize, offset, scale);
result char *buf; int bufsize, offset, scale;
```

5. Descriptors

Each process has access to resources through *descriptors*. Each descriptor is a handle allowing the process to reference objects such as files, devices and communications links.

5.1. The Reference Table

Rather than allowing processes direct access to descriptors, the system introduces a level of indirection, so that descriptors may be shared between processes. Each process has a *descriptor reference table*, containing pointers to the actual descriptors. The descriptors themselves thus have multiple references, and are reference counted by the system.

Each process has a fixed size descriptor reference table, where the size is returned by the *getdtablesize* call:

```
nds = getdtablesize();
result int nds;
```

and guaranteed to be at least as large as the constant NOFILE defined in <*param.h*>. The entries in the descriptor reference table are referred to by small integers; for example if there are 20 slots they are numbered 0 to 19.

5.2. Descriptor Properties

Each descriptor has a logical set of properties maintained by the system and defined by its *type*. Each type supports a set of operations; some operations, such as reading and writing, are common to several abstractions, while others are unique. The generic operations applying to many of these types are described in section 8. Naming contexts, files and directories are described in section 9. Section 10 describes communications domains and sockets. Terminals and (structured and unstructured) devices are described in section 11.

5.3. Managing Descriptor References

A duplicate of a descriptor reference may be made by doing

```
new = dup(old);
result int new; int old;
```

returning a copy of descriptor reference *old* indistinguishable from the original. The *new* chosen by the system will be the smallest unused descriptor reference slot. A copy of a descriptor reference may be made in a specific slot by doing

```
dup2(old, new);
int old, new;
```

The *dup2* call causes the system to deallocate the descriptor reference current occupying slot *new*, if any, replacing it with a reference to the same descriptor as old. This deallocation is also performed by:

```
close(old);
int old;
```

5.4. Multiplexing Requests

The system provides a standard way to do synchronous and asynchronous multiplexing of operations.

Synchronous multiplexing is performed by using the *select* call:

```
nds = select(nd, in, out, except, tvp);
result int nds; int nd; result *in, *out, *except;
struct timeval *tvp;
```

The *select* call examines the descriptors specified by the sets *in*, *out* and *except*, replacing the specified bit masks by the subsets that select for input, output, and exceptional conditions respectively (*nd* indicates the size, in bits, of the bit masks). If any descriptors meet the following criteria, then the number of such descriptors is returned in *nds* and the bit masks are updated.

- A descriptor selects for input if an input oriented operation such as *read* or *receive* is possible, or if a connection request may be accepted (see section 10.1.4).
- A descriptor selects for output if an output oriented operation such as *write* or *send* is possible, or if an operation that was "in progress", such as connection establishment, has completed (see section 8.3).
- A descriptor selects for an exceptional condition if a condition that would cause a SIGURG signal to be generated exists (see section 3.1).

If none of the specified conditions is true, the operation blocks for at most the amount of time specified by *tvp*, or waits for one of the conditions to arise if *tvp* is given as 0.

Options affecting i/o on a descriptor may be read and set by the call:

```
dopt = fcntl(d, cmd, arg);
result int dopt; int d, cmd, arg;

/* interesting values for cmd */
#define F_SETFL     3     /* set descriptor options */
#define F_GETFL     4     /* get descriptor options */
#define F_SETOWN    5     /* set descriptor owner (pid/pgrp) */
#define F_GETOWN    6     /* get descriptor owner (pid/pgrp) */
```

The F_SETFL *cmd* may be used to set a descriptor in non-blocking i/o mode and/or enable signalling when i/o is possible. F_SETOWN may be used to specify a process or process group to be signalled when using the latter mode of operation.

Operations on non-blocking descriptors will either complete immediately, note an error EWOULDBLOCK, partially complete an input or output operation returning a partial count, or return an error EINPROGRESS noting that the requested operation is in progress. A descriptor which has signalling enabled will cause the specified process and/or process group be signaled, with a SIGIO for input, output, or in-progress operation complete, or a SIGURG for exceptional conditions.

For example, when writing to a terminal using non-blocking output, the system will accept only as much data as there is buffer space for and return; when making a connection on a *socket*, the operation may return indicating that the connection establishment is "in progress". The *select* facility can be used to determine when further output is possible on the terminal, or when the connection establishment attempt is complete.

6. Resource Controls

6.1. Process Priorities

The system gives CPU scheduling priority to processes that have not used CPU time recently. This tends to favor interactive processes and processes that execute only for short periods. It is possible to determine the priority currently assigned to a process, process group, or the processes of a specified user, or to alter this priority using the calls:

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

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

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

The value *prio* is in the range −20 to 20. The default priority is 0; lower priorities cause more favorable execution. 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 super-user may lower priorities.

6.2. Resource Utilization

The resources used by a process are returned by a *getrusage* call, returning information in a structure defined in <sys/resource.h>:

```
#define RUSAGE_SELF     0       /* usage by this process */
#define RUSAGE_CHILDREN -1/* usage by all children */

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

struct rusage {
        struct  timeval ru_utime; /* user time used */
        struct  timeval ru_stime; /* system time used */
        int     ru_maxrss;        /* maximum core resident set size: kbytes */
        int     ru_ixrss;         /* integral shared memory size (kbytes*sec) */
        int     ru_idrss;         /* unshared data " */
        int     ru_isrss;         /* unshared stack " */
        int     ru_minflt;        /* page-reclaims */
        int     ru_majflt;        /* page faults */
        int     ru_nswap;         /* swaps */
        int     ru_inblock;       /* block input operations */
        int     ru_oublock;       /* block output " */
        int     ru_msgsnd;        /* messages sent */
        int     ru_msgrcv;        /* messages received */
        int     ru_nsignals;      /* signals received */
```

```
        int         ru_nvcsw;        /* voluntary context switches */
        int         ru_nivcsw;       /* involuntary " */
};
```

The *who* parameter specifies whose resource usage is to be returned. The resources used by the current process, or by all the terminated children of the current process may be requested.

6.3. Resource Limits

The resources of a process for which limits are controlled by the kernel are defined in <sys/resource.h>, and controlled by the *getrlimit* and *setrlimit* calls:

```
#define RLIMIT_CPU       0     /* cpu time in milliseconds */
#define RLIMIT_FSIZE     1     /* maximum file size */
#define RLIMIT_DATA      2     /* maximum data segment size */
#define RLIMIT_STACK     3     /* maximum stack segment size */
#define RLIMIT_CORE      4     /* maximum core file size */
#define RLIMIT_RSS       5     /* maximum resident set size */

#define RLIM_NLIMITS     6

#define RLIM_INFINITY    0x7fffffff

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

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

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

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

7. System operation support

The calls in this section are permitted only to a privileged user.

7.1. Bootstrap Operations

The call

```
mount(blkdev, dir, ronly);
char *blkdev, *dir; int ronly;
```

extends the UNIX name space. The *mount* call specifies a block device *blkdev* containing a UNIX file system to be made available starting at *dir*. If *ronly* is set then the file system is read-only; writes to the file system will not be permitted and access times will not be updated when files are referenced.

The call

```
swapon(blkdev, size);
char *blkdev; int size;
```

specifies a device to be made available for paging and swapping.

7.2. Shutdown Operations

The call

```
unmount(dir);
char *dir;
```

unmounts the file system mounted on *dir*. This call will succeed only if the file system is not currently being used.

The call

```
sync();
```

schedules input/output to clean all system buffer caches.

The call

```
reboot(how);
int how;
```

causes a machine halt or reboot. The call may request a reboot by specifying *how* as RB_AUTOBOOT, or that the machine be halted with RB_HALT. These constants are defined in <sys/reboot.h>.

7.3. Accounting

The system optionally keeps an accounting record in a file for each process that exits on the system. The format of this record is beyond the scope of this document. The accounting may be enabled to a file *name* by doing

```
acct(path);
char *path;
```

If *path* is null, then accounting is disabled. Otherwise, the named file becomes the accounting file.

Part II — System Facilities

This part of the document discusses the system facilities that are not considered part of the kernel.

The system abstractions described are:

Directory Contexts
: A directory context is a position in the UNIX file system name space. Operations on files and other named objects in a file system are always specified relative to such a context.

Files
: Files are used to store uninterpreted sequence of bytes on which random access *reads* and *writes* may occur. Pages from files or devices may also be mapped into process address space. A directory may be read as a file†.

Communications Domains
: A communications domain represents an interprocess communications environment, such as the communications facilities of the UNIX system, communications in the INTERNET, or the resource sharing protocols and access rights of a resource sharing system on a local network.

Sockets
: A socket is an endpoint of communication and the focal point for IPC in a communications domain. Sockets may be created in pairs, or given names and used to rendezvous with other sockets in a communications domain, accepting connections from these sockets or exchanging messages with them. These operations model a labeled or unlabeled communications graph, and can be used in a wide variety of communications domains. Sockets can have different *types* to provide different semantics of communication, increasing the flexibility of the model.

Terminals and other devices
: Devices include terminals, providing input editing and interrupt generation and output flow control and editing, magnetic tapes, disks and other peripherals. They often support the generic *read* and *write* operations as well as a number of *ioctls*.

Processes
: Process descriptors provide facilities for control and debugging of other processes.

† Support for mapping files is not included in this release.

8. Generic Operations

Many system abstractions support the operations *read, write* and *ioctl.* We describe the basics of these common primitives here. Similarly, the mechanisms whereby normally synchronous operations may occur in a non-blocking or asynchronous fashion are common to all system-defined abstractions and are described here.

8.1. Read and Write

The *read* and *write* system calls can be applied to communications channels, files, terminals and devices. They have the form:

```
cc = read(fd, buf, nbytes);
result int cc; int fd; result caddr_t buf; int nbytes;

cc = write(fd, buf, nbytes);
result int cc; int fd; caddr_t buf; int nbytes;
```

The *read* call transfers as much data as possible from the object defined by *fd* to the buffer at address *buf* of size *nbytes.* The number of bytes transferred is returned in *cc,* which is -1 if a return occurred before any data was transferred because of an error or use of non-blocking operations.

The *write* call transfers data from the buffer to the object defined by *fd.* Depending on the type of *fd,* it is possible that the *write* call will accept some portion of the provided bytes; the user should resubmit the other bytes in a later request in this case. Error returns because of interrupted or otherwise incomplete operations are possible.

Scattering of data on input or gathering of data for output is also possible using an array of input/output vector descriptors. The type for the descriptors is defined in <sys/uio.h> as:

```
struct iovec {
        caddr_t    iov_msg;        /* base of a component */
        int        iov_len;        /* length of a component */
};
```

The calls using an array of descriptors are:

```
cc = readv(fd, iov, iovlen);
result int cc; int fd; struct iovec *iov; int iovlen;

cc = writev(fd, iov, iovlen);
result int cc; int fd; struct iovec *iov; int iovlen;
```

Here *iovlen* is the count of elements in the *iov* array.

8.2. Input/Output Control

Control operations on an object are performed by the *ioctl* operation:

```
ioctl(fd, request, buffer);
int fd, request; caddr_t buffer;
```

This operation causes the specified *request* to be performed on the object *fd.* The *request* parameter specifies whether the argument buffer is to be read, written, read and written, or is not needed, and also the size of the buffer, as well as the request. Different descriptor types and subtypes within descriptor types may use distinct *ioctl* requests. For example, operations on terminals control flushing of input and output queues and setting of terminal parameters; operations on disks cause formatting operations to occur; operations on tapes control tape positioning.

The names for basic control operations are defined in <sys/ioctl.h>.

8.3. Non-Blocking and Asynchronous Operations

A process that wishes to do non-blocking operations on one of its descriptors sets the descriptor in non-blocking mode as described in section 5.4. Thereafter the *read* call will return a specific EWOULDBLOCK error indication if there is no data to be *read*. The process may *select* the associated descriptor to determine when a read is possible.

Output attempted when a descriptor can accept less than is requested will either accept some of the provided data, returning a shorter than normal length, or return an error indicating that the operation would block. More output can be performed as soon as a *select* call indicates the object is writeable.

Operations other than data input or output may be performed on a descriptor in a non-blocking fashion. These operations will return with a characteristic error indicating that they are in progress if they cannot return immediately. The descriptor may then be *select*ed for *write* to find out when the operation can be retried. When *select* indicates the descriptor is writeable, a respecification of the original operation will return the result of the operation.

9. File System

The file system abstraction provides access to a hierarchical file system structure. The file system contains directories (each of which may contain other sub-directories) as well as files and references to other objects such as devices and inter-process communications sockets.

Each file is organized as a linear array of bytes. No record boundaries or system related information is present in a file. Files may be read and written in a random-access fashion. The user may read the data in a directory as though it were an ordinary file to determine the names of the contained files, but only the system may write into the directories. The file system stores only a small amount of ownership, protection and usage information with a file.

9.1. Naming

The file system calls take *path name* arguments. These consist of a zero or more component *file names* separated by "/" characters, where each file name is up to 255 ASCII characters excluding null and "/".

Each process always has two naming contexts: one for the root directory of the file system and one for the current working directory. These are used by the system in the filename translation process. If a path name begins with a "/", it is called a full path name and interpreted relative to the root directory context. If the path name does not begin with a "/" it is called a relative path name and interpreted relative to the current directory context.

The system limits the total length of a path name to 1024 characters.

The file name ".." in each directory refers to the parent directory of that directory.

The calls

```
chdir(path);
char *path;

chroot(path);
char *path;
```

change the current working directory and root directory context of a process. Only the super-user can change the root directory context of a process.

9.2. Creation and Removal

The file system allows directories, files and special devices, to be created and removed from the file system.

9.2.1. Directory Creation and Removal

A directory is created with the *mkdir* system call:

```
mkdir(path, mode);
char *path; int mode;
```

and removed with the *rmdir* system call:

```
rmdir(path);
char *path;
```

A directory must be empty if it is to be deleted.

9.2.2. File Creation

Files are created with the *open* system call,

```
fd = open(path, oflag, mode);
result int fd; char *path; int oflag, mode;
```

The *path* parameter specifies the name of the file to be created. The *oflag* parameter must include O_CREAT from below to cause the file to be created. The protection for the new file is specified in *mode*. Bits for *oflag* are defined in <sys/file.h>:

```
#define O_RDONLY     000    /* open for reading */
#define O_WRONLY     001    /* open for writing */
#define O_RDWR       002    /* open for read & write */
#define O_NDELAY     004    /* non-blocking open */
#define O_APPEND     010    /* append on each write */
#define O_CREAT      01000  /* open with file create */
#define O_TRUNC      02000  /* open with truncation */
#define O_EXCL       04000  /* error on create if file exists */
```

One of O_RDONLY, O_WRONLY and O_RDWR should be specified, indicating what types of operations are desired to be performed on the open file. The operations will be checked against the user's access rights to the file before allowing the *open* to succeed. Specifying O_APPEND causes writes to automatically append to the file. The flag O_CREAT causes the file to be created if it does not exist, with the specified *mode*, owned by the current user and the group of the containing directory.

If the open specifies to create the file with O_EXCL and the file already exists, then the *open* will fail without affecting the file in any way. This provides a simple exclusive access facility.

9.2.3. Creating References to Devices

The file system allows entries which reference peripheral devices. Peripherals are distinguished as *block* or *character* devices according by their ability to support block-oriented operations. Devices are identified by their "major" and "minor" device numbers. The major device number determines the kind of peripheral it is, while the minor device number indicates one of possibly many peripherals of that kind. Structured devices have all operations performed internally in "block" quantities while unstructured devices often have a number of special *ioctl* operations, and may have input and output performed in large units. The *mknod* call creates special entries:

```
mknod(path, mode, dev);
char *path; int mode, dev;
```

where *mode* is formed from the object type and access permissions. The parameter *dev* is a configuration dependent parameter used to identify specific character or block i/o devices.

9.2.4. File and Device Removal

A reference to a file or special device may be removed with the *unlink* call,

```
unlink(path);
char *path;
```

The caller must have write access to the directory in which the file is located for this call to be successful.

9.3. Reading and Modifying File Attributes

Detailed information about the attributes of a file may be obtained with the calls:

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

stat(path, stb);
char *path; result struct stat *stb;

fstat(fd, stb);
int fd; result struct stat *stb;
```

The *stat* structure includes the file type, protection, ownership, access times, size, and a count of hard links. If the file is a symbolic link, then the status of the link itself (rather than the file the link references) may be found using the *lstat* call:

```
lstat(path, stb);
char *path; result struct stat *stb;
```

Newly created files are assigned the user id of the process that created it and the group id of the directory in which it was created. The ownership of a file may be changed by either of the calls

```
chown(path, owner, group);
char *path; int owner, group;

fchown(fd, owner, group);
int fd, owner, group;
```

In addition to ownership, each file has three levels of access protection associated with it. These levels are owner relative, group relative, and global (all users and groups). Each level of access has separate indicators for read permission, write permission, and execute permission. The protection bits associated with a file may be set by either of the calls:

```
chmod(path, mode);
char *path; int mode;

fchmod(fd, mode);
int fd, mode;
```

where *mode* is a value indicating the new protection of the file. The file mode is a three digit octal number. Each digit encodes read access as 4, write access as 2 and execute access as 1, or'ed together. The 0700 bits describe owner access, the 070 bits describe the access rights for processes in the same group as the file, and the 07 bits describe the access rights for other processes.

Three additional bits exist: the 04000 "set-user-id" bit can be set on an executable file to cause the effective user-id of a process which executes the file to be set to the owner of that file; the 02000 bit has a similar effect on the effective group-id. The 01000 bit causes an image of an executable program to be saved longer than would otherwise be normal; this "sticky" bit is a hint to the system that a program is heavily used.

Finally, the access and modify times on a file may be set by the call:

```
utimes(path, tvp);
char *path; struct timeval *tvp[2];
```

This is particularly useful when moving files between media, to preserve relationships between the times the file was modified.

9.4. Links and Renaming

Links allow multiple names for a file to exist. Links exist independently of the file linked to.

Two types of links exist, *hard* links and *symbolic* links. A hard link is a reference counting mechanism that allows a file to have multiple names within the same file system. Symbolic links cause string substitution during the pathname interpretation process.

Hard links and symbolic links have different properties. A hard link insures the target file will always be accessible, even after its original directory entry is removed; no such guarantee exists for a symbolic link. Symbolic links can span file systems boundaries.

The following calls create a new link, named *path2*, to *path1*:

```
link(path1, path2);
char *path1, *path2;

symlink(path1, path2);
char *path1, *path2;
```

The *unlink* primitive may be used to remove either type of link.

If a file is a symbolic link, the "value" of the link may be read with the *readlink* call,

```
len = readlink(path, buf, bufsize);
result int len; result char *path, *buf; int bufsize;
```

This call returns, in *buf*, the null-terminated string substituted into pathnames passing through *path*.

Atomic renaming of file system resident objects is possible with the *rename* call:

```
rename(oldname, newname);
char *oldname, *newname;
```

where both *oldname* and *newname* must be in the same file system. If *newname* exists and is a directory, then it must be empty.

9.5. Extension and Truncation

Files are created with zero length and may be extended simply by writing or appending to them. While a file is open the system maintains a pointer into the file indicating the current location in the file associated with the descriptor. This pointer may be moved about in the file in a random access fashion. To set the current offset into a file, the *lseek* call may be used,

```
oldoffset = lseek(fd, offset, type);
result off_t oldoffset; int fd; off_t offset; int type;
```

where *type* is given in <sys/file.h> as one of,

```
#define L_SET     0   /* set absolute file offset */
#define L_INCR    1   /* set file offset relative to current position */
#define L_XTND    2   /* set offset relative to end-of-file */
```

The call "lseek(fd, 0, L_INCR)" returns the current offset into the file.

Files may have "holes" in them. Holes are void areas in the linear extent of the file where data has never been written. These may be created by seeking to a location in a file past the current end-of-file and writing. Holes are treated by the system as zero valued bytes.

A file may be truncated with either of the calls:

```
truncate(path, length);
char *path; int length;

ftruncate(fd, length);
int fd, length;
```

reducing the size of the specified file to *length* bytes.

9.6. Checking Accessibility

A process running with different real and effective user ids may interrogate the accessibility of a file to the real user by using the *access* call:

```
accessible = access(path, how);
result int accessible; char *path; int how;
```

Here *how* is constructed by or'ing the following bits, defined in <sys/file.h>:

```
#define F_OK   0   /* file exists */
#define X_OK   1   /* file is executable */
#define W_OK   2   /* file is writable */
#define R_OK   4   /* file is readable */
```

The presence or absence of advisory locks does not affect the result of *access*.

9.7. Locking

The file system provides basic facilities that allow cooperating processes to synchronize their access to shared files. A process may place an advisory *read* or *write* lock on a file, so that other cooperating processes may avoid interfering with the process' access. This simple mechanism provides locking with file granularity. More granular locking can be built using the IPC facilities to provide a lock manager. The system does not force processes to obey the locks; they are of an advisory nature only.

Locking is performed after an *open* call by applying the *flock* primitive,

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

where the *how* parameter is formed from bits defined in <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 */
```

Successive lock calls may be used to increase or decrease the level of locking. If an object is currently locked by another process when a *flock* call is made, the caller will be blocked until the current lock owner releases the lock; this may be avoided by including LOCK_NB in the *how* parameter. Specifying LOCK_UN removes all locks associated with the descriptor. Advisory locks held by a process are automatically deleted when the process terminates.

9.8. Disk Quotas

As an optional facility, each file system may be requested to impose limits on a user's disk usage. Two quantities are limited: the total amount of disk space which a user may allocate in a file system and the total number of files a user may create in a file system. Quotas are expressed as *hard* limits and *soft* limits. A hard limit is always imposed; if a user would exceed a hard limit, the operation which caused the resource request will fail. A soft limit results in the user receiving a warning message, but with allocation succeeding. Facilities are provided to turn soft limits into hard limits if a user has exceeded a soft limit for an unreasonable period of time.

To enable disk quotas on a file system the *setquota* call is used:

```
setquota(special, file);
char *special, *file;
```

where *special* refers to a structured device file where a mounted file system exists, and *file* refers to a disk quota file (residing on the file system associated with *special*) from which user quotas should be obtained. The format of the disk quota file is implementation dependent.

To manipulate disk quotas the *quota* call is provided:

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

quota(cmd, uid, arg, addr);
int cmd, uid, arg; caddr_t addr;
```

The indicated *cmd* is applied to the user ID *uid*. The parameters *arg* and *addr* are command specific. The file <sys/quota.h> contains definitions pertinent to the use of this call.

10. Interprocess Communications

10.1. Interprocess Communication Primitives

10.1.1. Communication Domains

The system provides access to an extensible set of communication *domains*. A communication domain is identified by a manifest constant defined in the file <sys/socket.h>. Important standard domains supported by the system are the UNIX domain, AF_UNIX, for communication within the system, and the "internet" domain for communication in the DARPA internet, AF_INET. Other domains can be added to the system.

10.1.2. Socket Types and Protocols

Within a domain, communication takes place between communication endpoints known as *sockets*. Each socket has the potential to exchange information with other sockets within the domain.

Each socket has an associated abstract type, which describes the semantics of communication using that socket. Properties such as reliability, ordering, and prevention of duplication of messages are determined by the type. The basic set of socket types is defined in <sys/socket.h>:

```
/* Standard socket types */
#define SOCK_DGRAM      1    /* datagram */
#define SOCK_STREAM     2    /* virtual circuit */
#define SOCK_RAW        3    /* raw socket */
#define SOCK_RDM        4    /* reliably-delivered message */
#define SOCK_SEQPACKET  5    /* sequenced packets */
```

The SOCK_DGRAM type models the semantics of datagrams in network communication: messages may be lost or duplicated and may arrive out-of-order. The SOCK_RDM type models the semantics of reliable datagrams: messages arrive unduplicated and in-order, the sender is notified if messages are lost. The *send* and *receive* operations (described below) generate reliable/unreliable datagrams. The SOCK_STREAM type models connection-based virtual circuits: two-way byte streams with no record boundaries. The SOCK_SEQPACKET type models a connection-based, full-duplex, reliable, sequenced packet exchange; the sender is notified if messages are lost, and messages are never duplicated or presented out-of-order. Users of the last two abstractions may use the facilities for out-of-band transmission to send out-of-band data.

SOCK_RAW is used for unprocessed access to internal network layers and interfaces; it has no specific semantics.

Other socket types can be defined.†

Each socket may have a concrete *protocol* associated with it. This protocol is used within the domain to provide the semantics required by the socket type. For example, within the

† This release does not support the SOCK_RDM and SOCK_SEQPACKET types.

"internet" domain, the SOCK_DGRAM type may be implemented by the UDP user datagram protocol, and the SOCK_STREAM type may be implemented by the TCP transmission control protocol, while no standard protocols to provide SOCK_RDM or SOCK_SEQPACKET sockets exist.

10.1.3. Socket Creation, Naming, and Service Establishment

Sockets may be *connected* or *unconnected*. An unconnected socket descriptor is obtained by the *socket* call:

```
s = socket(domain, type, protocol);
result int s; int domain, type, protocol;
```

An unconnected socket descriptor may yield a connected socket descriptor in one of two ways: either by actively connecting to another socket, or by becoming associated with a name in the communications domain and *accepting* a connection from another socket.

To accept connections, a socket must first have a binding to a name within the communications domain. Such a binding is established by a *bind* call:

```
bind(s, name, namelen);
int s; char *name; int namelen;
```

A socket's bound name may be retrieved with a *getsockname* call:

```
getsockname(s, name, namelen);
int s; result caddr_t name; result int *namelen;
```

while the peer's name can be retrieved with *getpeername*:

```
getpeername(s, name, namelen);
int s; result caddr_t name; result int *namelen;
```

Domains may support sockets with several names.

10.1.4. Accepting Connections

Once a binding is made, it is possible to *listen* for connections:

```
listen(s, backlog);
int s, backlog;
```

The *backlog* specifies the maximum count of connections that can be simultaneously queued awaiting acceptance.

An *accept* call:

```
t = accept(s, name, anamelen);
result int t; int s; result caddr_t name; result int *anamelen;
```

returns a descriptor for a new, connected, socket from the queue of pending connections on *s*.

10.1.5. Making Connections

An active connection to a named socket is made by the *connect* call:

```
connect(s, name, namelen);
int s; caddr_t name; int namelen;
```

It is also possible to create connected pairs of sockets without using the domain's name space to rendezvous; this is done with the *socketpair* call†:

```
socketpair(d, type, protocol, sv);
int d, type, protocol; result int sv[2];
```

Here the returned *sv* descriptors correspond to those obtained with *accept* and *connect*.

The call

```
pipe(pv);
result int pv[2];
```

creates a pair of SOCK_STREAM sockets in the UNIX domain, with pv[0] only writeable and pv[1] only readable.

10.1.6. Sending and Receiving Data

Messages may be sent from a socket by:

```
cc = sendto(s, buf, len, flags, to, tolen);
result int cc; int s; caddr_t buf; int len, flags; caddr_t to; int tolen;
```

if the socket is not connected or:

```
cc = send(s, buf, len, flags);
result int cc; int s; caddr_t buf; int len, flags;
```

if the socket is connected. The corresponding receive primitives are:

```
msglen = recvfrom(s, buf, len, flags, from, fromlenaddr);
result int msglen; int s; result caddr_t buf; int len, flags;
result caddr_t from; result int *fromlenaddr;
```

and

```
msglen = recv(s, buf, len, flags);
result int msglen; int s; result caddr_t buf; int len, flags;
```

In the unconnected case, the parameters *to* and *tolen* specify the destination or source of the message, while the *from* parameter stores the source of the message, and **fromlenaddr* initially gives the size of the *from* buffer and is updated to reflect the true length of the *from* address.

All calls cause the message to be received in or sent from the message buffer of length *len* bytes, starting at address *buf*. The *flags* specify peeking at a message without reading it or sending or receiving high-priority out-of-band messages, as follows:

---

† This release supports *socketpair* creation only in the "unix" communication domain.

```
#define MSG_PEEK        0x1    /* peek at incoming message */
#define MSG_OOB         0x2    /* process out-of-band data */
```

10.1.7. Scatter/Gather and Exchanging Access Rights

It is possible to scatter and gather data and to exchange access rights with messages. When either of these operations is involved, the number of parameters to the call becomes large. Thus the system defines a message header structure, in <sys/socket.h>, which is used to contain the parameters to the calls:

```
struct msghdr {
        caddr_t   msg_name;          /* optional address */
        int       msg_namelen;       /* size of address */
        struct    iov *msg_iov;      /* scatter/gather array */
        int       msg_iovlen;        /* # elements in msg_iov */
        caddr_t   msg_accrights;     /* access rights sent/received */
        int       msg_accrightslen;  /* size of msg_accrights */
};
```

Here *msg_name* and *msg_namelen* specify the source or 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 section 8.3. Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*. In the "unix" domain these are an array of integer descriptors, taken from the sending process and duplicated in the receiver.

This structure is used in the operations *sendmsg* and *recvmsg*:

```
sendmsg(s, msg, flags);
int s; struct msghdr *msg; int flags;

msglen = recvmsg(s, msg, flags);
result int msglen; int s; result struct msghdr *msg; int flags;
```

10.1.8. Using Read and Write with Sockets

The normal UNIX *read* and *write* calls may be applied to connected sockets and translated into *send* and *receive* calls from or to a single area of memory and discarding any rights received. A process may operate on a virtual circuit socket, a terminal or a file with blocking or non-blocking input/output operations without distinguishing the descriptor type.

10.1.9. Shutting Down Halves of Full-Duplex Connections

A process that has a full-duplex socket such as a virtual circuit and no longer wishes to read from or write to this socket can give the call:

```
shutdown(s, direction);
int s, direction;
```

where *direction* is 0 to not read further, 1 to not write further, or 2 to completely shut the connection down.

10.1.10. Socket and Protocol Options

Sockets, and their underlying communication protocols, may support *options*. These options may be used to manipulate implementation specific or non-standard facilities. The *getsockopt* and *setsockopt* calls are used to control options:

```
getsockopt(s, level, optname, optval, optlen);
int s, level, optname; result caddr_t optval; result int *optlen;

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

The option *optname* is interpreted at the indicated protocol *level* for socket *s*. If a value is specified with *optval* and *optlen*, it is interpreted by the software operating at the specified *level*. The *level* SOL_SOCKET is reserved to indicate options maintained by the socket facilities. Other *level* values indicate a particular protocol which is to act on the option request; these values are normally interpreted as a "protocol number".

10.2. UNIX Domain

This section describes briefly the properties of the UNIX communications domain.

10.2.1. Types of Sockets

In the UNIX domain, the SOCK_STREAM abstraction provides pipe-like facilities, while SOCK_DGRAM provides datagrams – unreliable message-style communications.

10.2.2. Naming

Socket names are strings and may appear in the UNIX file system name space through portals†.

10.2.3. Access Rights Transmission

The ability to pass UNIX descriptors with messages in this domain allows migration of service within the system and allows user processes to be used in building system facilities.

10.3. INTERNET Domain

This section describes briefly how the INTERNET domain is mapped to the model described in this section. More information will be found in the *Networking Implementation Notes* in the System Internals Manual.

† The current implementation of the UNIX domain embeds bound sockets in the UNIX file system name space; this is a side effect of the implementation.

10.3.1. Socket Types and Protocols

SOCK_STREAM is supported by the INTERNET TCP protocol; SOCK_DGRAM by the UDP protocol. The SOCK_SEQPACKET has no direct INTERNET family analogue; a protocol based on one from the XEROX NS family and layered on top of IP could be implemented to fill this gap.

10.3.2. Socket Naming

Sockets in the INTERNET domain have names composed of the 32 bit internet address, and a 16 bit port number. Options may be used to provide source routing for the address, security options, or additional addresses for subnets of INTERNET for which the basic 32 bit addresses are insufficient.

10.3.3. Access Rights Transmission

No access rights transmission facilities are provided in the INTERNET domain.

10.3.4. Raw Access

The INTERNET domain allows the super-user access to the raw facilities of the various network interfaces and the various internal layers of the protocol implementation. This allows administrative and debugging functions to occur. These interfaces are modeled as SOCK_RAW sockets.

11. Devices

The system uses a collection of device-drivers to access attached peripherals. Such devices are grouped into two classes: structured devices on which block-oriented input/output operations occur, and unstructured devices (the rest).

11.1. Structured Devices

Structured devices include disk and tape drives, and are accessed through a system buffer-caching mechanism, which permits them to be accessed as ordinary files are, performing reads and writes as necessary to allow random-access.

The *mount* command in the system allows a structured device containing a file system volume to be accessed through the UNIX file system calls.

Tape drives also typically provide a structured interface, although this is rarely used.

11.2. Unstructured Devices

Unstructured devices are those devices which do not support a randomly accessed block structure.

Communications lines, raster plotters, normal magnetic tape access (in large or variable size blocks), and access to disk drives permitting large block transfers and special operations like disk formatting and labelling all use unstructured device interfaces.

The writing of devices for unstructured devices other than communications lines is described in the *Device Driver Manual* in the System Internals Manual.

12. Debugging Support

The *ptrace* facility of version 7 UNIX is provided in this release. Planned enhancements which would allow a descriptor-based process control facility have not been implemented.

Part III — Summary of Facilities

Appendix A. Summary of Facilities

A.1. Kernel Primitives

A.1.1. Process Naming and Protection

sethostid	set UNIX host id
gethostid	get UNIX host id
sethostname	set UNIX host name
gethostname	get UNIX host name
getpid	get process id
fork	create new process
exit	terminate a process
execve	execute a different process
getuid	get user id
geteuid	get effective user id
setreuid	set real and effective user id's
getgid	get accounting group id
getegid	get effective accounting group id
getgroups	get access group set
setregid	set real and effective group id's
setgroups	set access group set
getpgrp	get process group
setpgrp	set process group

A.1.2. Memory Management

<mman.h>	memory management definitions
sbrk	change data section size
sstk†	change stack section size

† Not supported in the 1.1 Sun release.

getpagesize	get memory page size
mmap†	map pages of memory
mremap†	remap pages in memory
munmap†	unmap memory
mprotect†	change protection of pages
madvise†	give memory management advice
mincore†	determine core residency of pages

A.1.3. Signals

<signal.h>	signal definitions
sigvec	set handler for signal
kill	send signal to process
killpgrp	send signal to process group
sigblock	block set of signals
sigsetmask	restore set of blocked signals
sigpause	wait for signals
sigstack	set software stack for signals

A.1.4. Timing and Statistics

<sys/time.h>	time-related definitions
gettimeofday	get current time and timezone
settimeofday	set current time and timezone
getitimer	read an interval timer
setitimer	get and set an interval timer
profil	profile process

A.1.5. Descriptors

getdtablesize	descriptor reference table size
dup	duplicate descriptor
dup2	duplicate to specified index
close	close descriptor
select	multiplex input/output
fcntl	control descriptor options

† Not supported in the 1.1 Sun release.

A.1.6. Resource Controls

<sys/resource.h>	resource-related definitions
getpriority	get process priority
setpriority	set process priority
getrusage	get resource usage
getrlimit	get resource limitations
setrlimit	set resource limitations

A.1.7. System Operation Support

mount	mount a device file system
swapon	add a swap device
umount	umount a file system
sync	flush system caches
reboot	reboot a machine
acct	specify accounting file

A.2. System Facilities

A.2.1. Generic Operations

read	read data
write	write data
<sys/uio.h>	scatter-gather related definitions
readv	scattered data input
writev	gathered data output
<sys/ioctl.h>	standard control operations
ioctl	device control operation

A.2.2. File System

Operations marked with a * exist in two forms: as shown, operating on a file name, and operating on a file descriptor, when the name is preceded with a "f".

<sys/file.h>	file system definitions
chdir	change directory
chroot	change root directory
mkdir	make a directory
rmdir	remove a directory
open	open a new or existing file
mknod	make a special file
unlink	remove a link
stat*	return status for a file

lstat	returned status of link
chown*	change owner
chmod*	change mode
utimes	change access/modify times
link	make a hard link
symlink	make a symbolic link
readlink	read contents of symbolic link
rename	change name of file
lseek	reposition within file
truncate*	truncate file
access	determine accessibility
flock	lock a file

A.2.3. Interprocess Communications

<sys/socket.h>	standard definitions
socket	create socket
bind	bind socket to name
getsockname	get socket name
listen	allow queueing of connections
accept	accept a connection
connect	connect to peer socket
socketpair	create pair of connected sockets
sendto	send data to named socket
send	send data to connected socket
recvfrom	receive data on unconnected socket
recv	receive data on connected socket
sendmsg	send gathered data and/or rights
recvmsg	receive scattered data and/or rights
shutdown	partially close full-duplex connection
getsockopt	get socket option
setsockopt	set socket option

A.2.4. Devices

A.2.5. Debugging Support

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 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 super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

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 path name does not exist.

3 ESRCH No such process
The process whose number was given to *kill* and *ptrace* 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 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 core
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 a hardware fault 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, 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, for example in a path name 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: dismounting a non-mounted device, mentioning an unknown signal in *signal,* reading or writing a file for which *seek* has generated a negative pointer. 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 for writing 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 library (as described in section 3M) is out of the domain of the function.

34 ERANGE Result too large
The value of a function in the math library (as described in section 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 shouldn't 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
An request to send or receive data was disallowed because the socket is 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 *unused*

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 path name lookup involved more than 8 symbolic links.

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

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

DEFINITIONS

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.

Directory

A directory is a special type of file which contains entries which 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.

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's 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 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".

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 at the time 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, those users in the file's group, 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's effective user ID is that of the super-user.

The process's effective user ID matches the user ID of the owner of the file and the owner permissions allow the access.

The process's effective user ID does not match the user ID of the owner of the file, and either the process's effective group ID matches the group ID of the file, or the group ID of the file is in the process's group access list, and the group permissions allow the access.

Neither the effective user ID nor effective group ID and group access list of the process match the corresponding user ID and group ID of the file, but the permissions for "other users" allow access.

Otherwise, permission is denied.

File Name

Names consisting of up to 255 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 file names because of the special meaning attached to these characters by the shell.

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.

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.

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).

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.

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 real group ID. These are initialized from the equivalent attributes of the process which created it.

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's root directory need not be the root directory of the root file system.

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; consult *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.

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.

Super-user
: A process is recognized as a *super-user* process and is granted special privileges if its

effective user ID is 0.

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).

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 which 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*, is used to read and write data to and from the socket which connected to this one; it is not used to accept more connections. The original socket *s* remains open for accepting further connections.

The argument *addr* is a result parameter which 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 which 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 <sys/file.h>

#define R_OK    4    /* test for read permission */
#define W_OK    2    /* test for write permission */
#define X_OK    1    /* test for execute (search) permission */
#define F_OK    0    /* test for presence of file */

accessible = 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.

[ENOENT] The argument path name was too long.

[ENOENT] Read, write, or execute (search) permission is requested for a null path name or the named file does not exist.

[EPERM] The argument contains a byte with the high-order bit set.

[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.

SEE ALSO

chmod(2), stat(2)

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).

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. It is erroneous to try to turn on accounting when it is already on.

ERRORS

Acct will fail if one of the following is true:

[EPERM]	The caller is not the super-user.
[EPERM]	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

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 the *name*, be assigned to the socket.

NOTES

Binding a name in the UNIX domain creates a socket in the file system which 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.

SEE ALSO

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

BUGS

The file created is a side-effect of the current implementation and will not be created in future versions of the UNIX ipc domain.

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* 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.

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.

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.
[ENOENT]	The argument path name was too long.
[EPERM]	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.

SEE ALSO

chroot(2)

NAME

chmod, fchmod – change mode of file

SYNOPSIS

chmod(path, mode)
char *path;
int mode;

fchmod(fd, mode)
int fd, mode;

DESCRIPTION

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

04000 set user ID on execution
02000 set group ID on execution
01000 save text image after execution
00400 read by owner
00200 write by owner
00100 execute (search on directory) by owner
00070 read, write, execute (search) by group
00007 read, write, execute (search) by others

If an executable file is set up for sharing (this is the default) then mode 1000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Ability to set this bit is restricted to the super-user.

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:

[EPERM]	The argument contains a byte with the high-order bit set.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The pathname was too long.
[ENOENT]	The named file does not exist.
[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.

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

open(2), chown(2)

NAME

chown, fchown – change owner and group of a file

SYNOPSIS

```
chown(path, owner, group)
char *path;
int owner, group;

fchown(fd, owner, group)
int fd, owner, group;
```

DESCRIPTION

The file which is named by *path* or referenced by *fd* has its *owner* and *group* changed as specified. Only the super-user may execute this call, because if users were able to give files away, they could defeat the file-space accounting procedures.

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 (see *flock*(2)).

Only one of the owner and group id's may be set by specifying the other as –1.

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:

[EINVAL]	The argument path does not refer to a file.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The argument pathname is too long.
[EPERM]	The argument contains a byte with the high-order bit set.
[ENOENT]	The named file does not exist.
[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.

Fchown will fail if:

[EBADF]	*Fd* does not refer to a valid descriptor.
[EINVAL]	*Fd* refers to a socket, not a file.

SEE ALSO

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 "/". This root directory setting is inherited across *execve*(2) and by all children of this process created with *fork*(2) calls.

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.
[EPERM]	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

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 *seek* 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, see *flock*(2) for further information.

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 forks (see *fork*(2)), 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, 1)" 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.

Close unmaps pages mapped through this file 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.

SEE ALSO

accept(2), flock(2), open(2), pipe(2), socket(2), socketpair(2), execve(2), fcntl(2), mmap(2), munmap(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 permanently specifies the peer to which datagrams are to be sent; if it 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.

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.
[EWOULDBLOCK]	The socket is non-blocking and the and the connection cannot be completed immediately. It is possible to *select*(2) the socket while it is connecting by selecting it for writing.

SEE ALSO

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

NAME

creat – create a new file

SYNOPSIS

creat(name, mode)
char *name;

DESCRIPTION

This interface is obsoleted by open(2).

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 *umask*(2)). Also 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.

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) facilitity.

RETURN VALUE

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

ERRORS

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

[EPERM]	The argument contains a byte with the high-order bit set.
[ENOTDIR]	A component of the path prefix is not a directory.
[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.
[EISDIR]	The file is a directory.
[EMFILE]	There are already too many files open.
[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).

SEE ALSO

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

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 which 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.

SEE ALSO

accept(2), open(2), close(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. See *a.out*(5).

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, 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 which are not directly arguments to the command, see *environ*(5).

Descriptors open in the calling process remain open in the new process, except for those for which the close-on-exec flag is set; see *close*(2). 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 be undefined; see *sigvec*(2) for more information.

Each process has a *real* user ID and group ID and an *effective* user ID and group ID. 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 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)
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 *sigvec*(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*(5) 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:

[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.
[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)).
[E2BIG]	The number of bytes in the new process's argument list is larger than the system-imposed limit of {ARG_MAX} bytes.
[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.

CAVEATS

If a program is *setuid* to a non-super-user, but is executed when the real *uid* is "root", then the program has the powers of a super-user as well.

SEE ALSO

exit(2), fork(2), execl(3), environ(5)

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; see *wait*(2). The low-order 8 bits of *status* are available to the parent process.

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 (see *intro*(2)) inherits each of these processes as well.

Most C programs will 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

fcntl – file control

SYNOPSIS

```
#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 *execve*(2) system calls.

F_GETFD — Get the close-on-exec flag associated with the file descriptor *fd*. If the low-order bit is 0, the file will remain open across *exec*, otherwise the file will be closed upon execution of *exec*.

F_SETFD — Set the close-on-exec flag associated with *fd* to the low order bit of *arg* (0 or 1 as above).

F_GETFL — Get descriptor status flags, see *fcntl*(5) for their definitions.

F_SETFL — Set descriptor status flags, see *fcntl*(5) for their definitions.

F_GETOWN — Get the process ID or process group currently receiving SIGIO and SIGURG signals; process groups are returned as negative values.

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.

The SIGIO facilities are enabled by setting the FASYNC flag with F_SETFL.

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 the low-order bit is defined).
F_GETFL	Value of flags.
F_GETOWN	Value of file descriptor owner.
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:

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

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

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

SEE ALSO

close(2), execve(2), getdtablesize(2), open(2), sigvec(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 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 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.

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

pid = fork()
int pid;

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, so that, 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.

The child processes resource utilizations are set to 0; see *setrlimit*(2).

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.

SEE ALSO

execve(2), wait(2)

NAME

fsync – synchronize a file's in-core 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: all in-core modified copies of buffers for the associated file have been written to a disk when the call returns. (Note that this is different than *sync*(2) which schedules disk-io for all files (as though an *fsync* had been done on all files) but returns before the i/o completes.)

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), cron(8)

BUGS

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

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

getgid, getegid – get group identity

SYNOPSIS

gid = getgid()
int gid;

egid = getegid()
int egid;

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, and it is for such processes that *getgid* is most useful.

SEE ALSO

getuid(2), setregid(2), setgid(3C)

NAME

getgroups – get group access list

SYNOPSIS

#include <sys/param.h>

ngroups = getgroups(n, &gidset)
int ngroups;
int n, *gidset;

DESCRIPTION

Getgroups gets the current group access list of the user process and stores it in the array *gidset*. The parameter *n* indicates the number of entries which may be placed in *gidset* and *getgroups* returns the actual number of entries placed in the *gidset* array. No more than NGRPS, as defined in *<sys/param.h>*, will ever be returned.

RETURN VALUE

A return value of greater than zero indicates the number of entries placed in the *gidset* array. A return value of –1 indicates that an error occurred, and the error code is stored in the global variable *errno*.

ERRORS

The possible errors for *getgroup* are:

[EFAULT] The arguments *ngroups* or *gidset* specify invalid addresses.

SEE ALSO

setgroups(2), initgroups(3)

NAME

gethostid – get unique identifier of current host

SYNOPSIS

hostid = gethostid()
int hostid;

DESCRIPTION

Gethostid returns the 32-bit identifier for the current host, which is unique across all hosts.

SEE ALSO

hostid(1)

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 super-user 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 int the global location *errno*.

ERRORS

The following errors may be returned by these calls:

[EFAULT]	The *name* or *namelen* parameter gave an invalid address.
[EPERM]	The caller was not the super-user.

SEE ALSO

gethostid(2)

BUGS

Host names are limited to 255 characters.

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).

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 this resolution.

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 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

sbrk(2), pagesize(1)

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).

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

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

BUGS

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

NAME

getpgrp – get process group

SYNOPSIS

pgrp = getpgrp(pid)
int prgp;
int pid;

DESCRIPTION

The process group of the specified process is returned by *getpgrp*. If *pid* is zero, then the call applies to the current process.

Process groups are used for distribution of signals, and by terminals to arbitrate requests for their input: processes which have the same process group as the terminal are foreground and may read, while others will block with a signal if they attempt to read.

This call is thus used by programs such as *csh*(1) to create process groups in implementing job control. The TIOCGPGRP and TIOCSPGRP calls described in *tty*(4) are used to get/set the process group of the control terminal.

SEE ALSO

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

NAME

getpid, getppid – get process identification

SYNOPSIS

pid = getpid()
long pid;

ppid = getppid()
long ppid;

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.

SEE ALSO

gethostid(2)

NAME

getpriority, setpriority – get/set program scheduling priority

SYNOPSIS

```
#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 –20 to 20. The default priority is 0; lower priorities cause more favorable scheduling.

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 super-user 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 super-user attempted to change a process priority to a negative value.

SEE ALSO

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

BUGS

It is not possible for the process executing *setpriority ( )* to lower any other process down to its current priority, without requiring superuser privileges.

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 milliseconds) 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 automatically by the system.
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.

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 RLIMIT_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.

[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 about the resources utilized by the current process, or all its terminated child processes. The *who* parameter is one of RUSAGE_SELF or RUSAGE_CHILDREN. If *rusage* is non-zero, 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 */
        int     ru_maxrss;
        int     ru_ixrss;               /* integral shared memory size */
        int     ru_idrss;               /* integral unshared data size */
        int     ru_isrss;               /* integral unshared stack size */
        int     ru_minflt;              /* page reclaims */
        int     ru_majflt;              /* page faults */
        int     ru_nswap;               /* swaps */
        int     ru_inblock;             /* block input operations */
        int     ru_oublock;             /* block output operations */
        int     ru_msgsnd;              /* messages sent */
        int     ru_msgrcv;              /* messages received */
        int     ru_nsignals;            /* signals received */
        int     ru_nvcsw;               /* voluntary context switches */
        int     ru_nivcsw;              /* involuntary context switches */
};
```

The fields are interpreted as follows:

ru_utime the total amount of time spent executing in user mode. Time is given in seconds:microseconds.

ru_stime the total amount of time spent in the system executing on behalf of the process(es). Time is given in seconds:microseconds.

ru_maxrss the maximum resident set size utilized. Size is given in pages (1 page = 2Kbytes).

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 pages * clock ticks (1 tick = 1/50 second). The value is calculated by summing the number of shared memory pages in use each time the internal system clock ticks, and then averaging over 1 second intervals.

ru_idrss an integral value of the amount of unshared memory residing in the data segment of a process. The value is given in pages * clock ticks.

ru_isrss an integral value of the amount of unshared memory residing in the stack segment of a process. The value is given in pages * clock ticks.

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/o 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 input.
ru_outblock	the number of times the file system had to perform 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.

NOTES

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

SEE ALSO

gettimeofday(2), wait(2)

BUGS

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

NAME

getsockname – get socket name

SYNOPSIS

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), getpeername(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 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 to be supplied or 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; see *socket*(2). Options at other protocol levels vary in format and name, consult the appropriate entries in (4P).

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.
[EFAULT]	The options are not in a valid part of the process address space.

SEE ALSO

socket(2), getprotoent(3N)

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 in seconds and microseconds since midnight January 1, 1970.

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

```
struct timeval {
        u_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.

If *tp* and/or *tzp* is a zero pointer, the corresponding information will not be returned or set.

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)

BUGS

Time is never correct enough to believe the microsecond values. There should a mechanism by which, at least, local clusters of systems might synchronize their clocks to millisecond granularity.

NAME

getuid, geteuid – get user identity

SYNOPSIS

uid = getuid()
int uid;

euid = geteuid()
int euid;

DESCRIPTION

Getuid returns the real user ID of the current process, *geteuid* 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, which use *getuid* to determine the real-user-id of the process which invoked them.

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.

If no error has occurred (using a STANDARD device driver), a value of 0 is returned.

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.

SEE ALSO

execve(2), fcntl(2), mtio(4), tty(4)

NAME

kill – send signal to a process

SYNOPSIS

kill(pid, sig)
int pid, sig;

DESCRIPTION

Kill sends the signal *sig* to a process, specified by the process number *pid*. *Sig* may be one of the signals specified in *sigvec*(2), or it may be 0, in which case error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The sending and receiving processes must have the same effective user ID, otherwise this call is restricted to the super-user. A single exception is the signal SIGCONT which may always be sent to any child or grandchild of the current process.

If the process number is 0, the signal is sent to all other processes in the sender's 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.

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*.

[EPERM] The sending process is not the super-user and its effective user id does not match the effective user-id of the receiving process.

SEE ALSO

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 were found in the specified process group.

[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

link – make a hard link to a file

SYNOPSIS

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:

[EPERM]	Either pathname contains a byte with the high-order bit set.
[ENOENT]	Either pathname was too long.
[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 *name1* does not exist.
[EEXIST]	The link named by *name2* does exist.
[EPERM]	The file named by *name1* is a directory and the effective user ID is not super-user.
[EXDEV]	The link named by *name2* and the file named by *name1* are on different file systems.
[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]	One of the pathnames specified is outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

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 backlog for incoming connections is 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_PKTSTREAM.

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 will receive an error with an indication of ECONNREFUSED.

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, tell – move read/write pointer

SYNOPSIS

```
#define L_SET   0    /* set the seek pointer */
#define L_INCR  1    /* increment the seek pointer */
#define L_XTND  2    /* extend the file size */

pos = lseek(d, offset, whence)
int pos;
int d, offset, whence;
```

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 L_SET, the pointer is set to *offset* bytes.

If *whence* is L_INCR, the pointer is set to its current location plus *offset*.

If *whence* is L_XTND, 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.

The obsolete function *tell(fildes)* is identical to *lseek(fildes, 0L, L_INCR)*.

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.

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]	*Fildes* is not an open file descriptor.
[ESPIPE]	*Fildes* is associated with a pipe or a socket.
[EINVAL]	*Whence* is not a proper value.
[EINVAL]	The resulting file pointer would be negative.

SEE ALSO

dup(2), open(2)

NAME

mkdir – make a directory file

SYNOPSIS

mkdir(path, mode)
char *path;
int mode;

DESCRIPTION

Mkdir creates a new directory file with name *path*. The mode of the new file is initialized from *mode*. (The protection part of the mode is modified by the process's mode mask; see *umask*(2)).

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. See *umask*(2).

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:

[EPERM]	The process's effective user ID is not super-user.
[EPERM]	The *path* argument contains a byte with the high-order bit set.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EROFS]	The named file resides on a read-only file system.
[EEXIST]	The named file exists.
[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 occured while writing to the file system.

SEE ALSO

chmod(2), stat(2), umask(2)

NAME

mknod – make a special file

SYNOPSIS

mknod(path, mode, dev)
char *path;
int mode, dev;

DESCRIPTION

Mknod creates a new file whose name is *path*. The mode of the new file (including special file bits) is initialized from *mode*. (The protection part of the mode is modified by the process's mode mask; see *umask*(2)). The first block pointer of the i-node is initialized from *dev* and is used to specify which device the special file refers to.

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.

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 will fail and the file mode will be unchanged if:

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

SEE ALSO

chmod(2), stat(2), umask(2)

NAME

mmap – map pages of memory

SYNOPSIS

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

mmap(addr, len, prot, share, fd, off)
caddr_t addr; int len, prot, share, fd; off_t off;
```

DESCRIPTION

N.B.: This call is not completely implemented in 4.2.

Mmap maps the pages starting at *addr* and continuing for *len* bytes from the object represented by the descriptor *fd*, at the current file position of offset *off*. The parameter *share* specifies whether modifications made to this mapped copy of the page are to be kept *private* or are to be *shared* with other references. The parameter *prot* specifies the accessibility of the mapped pages. The *addr* and *len* parameters and the sum of the current position in *fd* and the *off* parameters must be multiples of the page size (found using the *getpagesize*(2) call).

Pages are automatically unmapped at *close*.

RETURN VALUE

The call returns 0 on success, –1 on failure.

ERRORS

The *mmap* call will fail if:

[EINVAL] The argument address or length is not a multiple of the page size as returned by *getpagesize*(2),*or* the length is negative.

[EINVAL] The entire range of pages specified in the call is not part of data space.

[EINVAL] The specified *fd* does not refer to a character special device which supports mapping (e.g. a frame buffer).

[EINVAL] The specified *fd* is not open for reading and read access is requested, or not open for writing when write access is requested.

[EINVAL] The sharing mode was not specified as MAP_SHARED.

SEE ALSO

getpagesize(2), munmap(2), close(2)

NAME

mount, umount – mount or remove file system

SYNOPSIS

mount(special, name, rwflag)
char *special, *name;
int rwflag;

umount(special)
char *special;

DESCRIPTION

Mount announces to the system that a removable file system has been mounted on the block-structured special file *special;* from now on, references to file *name* will refer to the root file on the newly mounted file system. *Special* and *name* are pointers to null-terminated strings containing the appropriate path names.

Name must exist already. *Name* must be a directory. Its old contents are inaccessible while the file system is mounted.

The *rwflag* argument determines whether the file system can be written on; if it is 0 writing is allowed, if non-zero no writing is done. 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.

Umount announces to the system that the *special* file is no longer to contain a removable file system. The associated file reverts to its ordinary interpretation.

RETURN VALUE

Mount returns 0 if the action occurred, –1 if *special* is inaccessible or not an appropriate file; if *name* does not exist; if *special* is already mounted; if *name* is in use; or if there are already too many file systems mounted.

Umount returns 0 if the action occurred; –1 if if the special file is inaccessible or does not have a mounted file system, or if there are active files in the mounted file system.

ERRORS

Mount will fail when one of the following occurs:

[NODEV]	The caller is not the super-user.
[NODEV]	*Special* does not exist.
[ENOTBLK]	*Special* is not a block device.
[ENXIO]	The major device number of *special* is out of range (this indicates no device driver exists for the associated hardware).
[EPERM]	The pathname contains a character with the high-order bit set.
[ENOTDIR]	A component of the path prefix in *name* is not a directory.
[EROFS]	*Name* resides on a read-only file system.
[EBUSY]	*Name* is not a directory, or another process currently holds a reference to it.
[EBUSY]	No space remains in the mount table.
[EBUSY]	The super block for the file system had a bad magic number or an out of range block size.
[EBUSY]	Not enough memory was available to read the cylinder group information for the file system.
[EBUSY]	An i/o error occurred while reading the super block or cylinder group information.

Umount may fail with one of the following errors:

[NODEV]	The caller is not the super-user.
[NODEV]	*Special* does not exist.
[ENOTBLK]	*Special* is not a block device.
[ENXIO]	The major device number of *special* is out of range (this indicates no device driver exists for the associated hardware).
[EINVAL]	The requested device is not in the mount table.
[EBUSY]	A process is holding a reference to a file located on the file system.

SEE ALSO

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

munmap – unmap pages of memory

SYNOPSIS

#include <mman.h>

munmap(addr, len)
caddr_t addr; int len;

DESCRIPTION

N.B.: This call is not completely implemented in 4.2.

Munmap causes the pages starting at *addr* and continuing for *len* bytes to refer to private pages which will be initialized to zero on reference.

RETURN VALUE

The call returns –1 on error, 0 on success.

ERRORS

The call fails if any of the following:

[EINVAL] The argument address or length is not a multiple of the page size as returned by *getpagesize*(2), *or* the length is negative.

[EINVAL] The entire range of pages specified in the call is not part of data space.

SEE ALSO

brk (2), mmap(2), close(2)

NAME

open – open a file for reading or writing, or create a new file

SYNOPSIS

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

open(path, flags, mode)
char *path;
int flags, mode;
```

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 mode *mode* as described in *chmod*(2) and modified by the process' umask value (see *umask*(2)).

Path is the address of a string of ASCII characters representing a path name, terminated by a null character. The flags specified are formed by *or*'ing the following values

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_NDELAY	do not block on open
O_APPEND	append on each write
O_CREAT	create file if it does not exist
O_TRUNC	truncate size to 0
O_EXCL	error if create and file exists

Opening a file with O_APPEND set causes each write on the file to be appended to the end. If O_TRUNC is specified and the file exists, the file is truncated to zero length. If O_EXCL is set with O_CREAT, then if the file already exists, the open returns an error. This can be used to implement a simple exclusive access locking mechanism. If the O_NDELAY flag is specified and the open call would result in the process being blocked for some reason (e.g. waiting for carrier on a dialup line), the open returns immediately. The first time the process attempts to perform i/o on the open file it will block (not currently implemented).

Upon successful completion a non-negative integer termed a file descriptor is returned. 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* system calls; see *close*(2).

There is a system enforced limit on the number of open file descriptors per process, whose value is 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:

[EPERM]	The pathname contains a character with the high-order bit set.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	O_CREAT is not set and the named file does not exist.
[EACCES]	A component of the path prefix denies search permission.
[EACCES]	The required permissions (for reading and/or writing) are denied for the named file.
[EISDIR]	The named file is a directory, and the arguments specify it is to be opened for writing.

[EROFS]	The named file resides on a read-only file system, and the file is to be modified.
[EMFILE]	{OPEN_MAX} file descriptors are currently open.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and the *open* call requests write access.
[EFAULT]	*Path* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EEXIST]	O_EXCL was specified and the file exists.
[ENXIO]	The O_NDELAY flag is given, and the file is a communications device on which there is no carrier present.
[EOPNOTSUPP]	An attempt was made to open a socket (not currently implemented).

SEE ALSO

chmod(2), close(2), dup(2), lseek(2), read(2), write(2), umask(2)

NAME

pipe – create an interprocess communication channel

SYNOPSIS

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* calls) will pass data through the pipe with *read* and *write* calls.

The shell has a syntax to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors closed) returns an end-of-file.

Pipes are really a special case of the *socketpair*(2) call and, in fact, are implemented as such in the system.

A signal is generated if a write on a pipe with only one end is attempted.

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)

BUGS

Should more than 4096 bytes be necessary in any pipe among a loop of processes, deadlock will occur.

NAME

profil – execution time profile

SYNOPSIS

profil(buff, bufsiz, offset, scale)
char *buff;
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 (20 milliseconds); *offset* is subtracted from it, and the result multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0x10000 gives a 1-1 mapping of pc's to words in *buff;* 0x8000 maps each pair of instruction words together. 0x2 maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

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

gprof(1), setitimer(2), monitor(3)

NAME
ptrace – process trace

SYNOPSIS
#include <signal.h>

ptrace(request, pid, addr, data)
int request, pid, *addr, data;

DESCRIPTION
Ptrace provides a means by which a parent process may control the execution of a child process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging. There are four arguments whose interpretation depends on a *request* argument. Generally, *pid* is the process ID of the traced process, which must be a child (no more distant descendant) of the tracing process. A process being traced behaves normally until it encounters some signal whether internally generated like "illegal instruction" or externally generated like "interrupt". See *sigvec*(2) for the list. Then the traced process enters a stopped state and its parent is notified via *wait*(2). When the child is in the stopped state, its core image can be examined and modified using *ptrace*. If desired, another *ptrace* request can then cause the child either to terminate or to continue, possibly ignoring the signal.

The value of the *request* argument determines the precise action of the call:

0 This request is the only one used by the child process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

1,2 The word in the child process's address space at *addr* is returned. If I and D space are separated (e.g. historically on a pdp-11), request 1 indicates I space, 2 D space. *Addr* must be even. The child must be stopped. The input *data* is ignored.

3 The word of the system's per-process data area corresponding to *addr* is returned. *Addr* must be even and less than 512. This space contains the registers and other information about the process; its layout corresponds to the *user* structure in the system.

4,5 The given *data* is written at the word in the process's address space corresponding to *addr*, which must be even. No useful value is returned. If I and D space are separated, request 4 indicates I space, 5 D space. Attempts to write in pure procedure fail if another process is executing the same file.

6 The process's system data is written, as it is read with request 3. Only a few locations can be written in this way: the general registers, the floating point status and registers, and certain bits of the processor status word.

7 The *data* argument is taken as a signal number and the child's execution continues at location *addr* as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal that caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop. If *addr* is (int *)1 then execution continues from where it stopped.

8 The traced process terminates.

9 Execution continues as in request 7; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. (On the Sun and VAX-11 the T-bit is used and just one instruction is executed.) This is part of the mechanism for implementing breakpoints.

As indicated, these calls (except for request 0) can be used only when the subject process has stopped. The *wait* call is used to determine when a process stops; in such a case the "termination" status returned by *wait* has the value 0177 to indicate stoppage rather than genuine termination.

To forestall possible fraud, *ptrace* inhibits the set-user-id and set-group-id facilities on subsequent *execve*(2) calls. If a traced process calls *execve*, it will stop before executing the first instruction of the new image showing signal SIGTRAP.

On the Sun and VAX-11, "word" also means a 32-bit integer, but the "even" restriction does not apply.

RETURN VALUE

A 0 value is returned if the call succeeds. If the call fails then a -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL]	The request code is invalid.
[EINVAL]	The specified process does not exist.
[EINVAL]	The given signal number is invalid.
[EFAULT]	The specified address is out of bounds.
[EPERM]	The specified process cannot be traced.

SEE ALSO

wait(2), sigvec(2), adb(1S)

BUGS

Ptrace is unique and arcane; it should be replaced with a special file which can be opened and read and written. The control functions could then be implemented with *ioctl*(2) calls on this file. This would be simpler to understand and have much higher performance.

The request 0 call should be able to specify signals which are to be treated normally and not cause a stop. In this way, for example, programs with simulated floating point (which use "illegal instruction" signals at a very high rate) could be efficiently debugged.

The error indication, -1, is a legitimate function value; *errno*, see *intro*(2), can be used to disambiguate.

It should be possible to stop a process on occurrence of a system call; in this way a completely controlled environment could be provided.

NAME

quota – manipulate disk quotas

SYNOPSIS

#include <sys/quota.h>

quota(cmd, uid, arg, addr)
int cmd, uid, arg;
caddr_t addr;

DESCRIPTION

The *quota* call manipulates disk quotas for file systems which have had quotas enabled with *setquota*(2). The *cmd* parameter indicates a command to be applied to the user ID *uid*. *Arg* is a command specific argument and *addr* is the address of an optional, command specific, data structure which is copied in or out of the system. The interpretation of *arg* and *addr* is given with each command below.

Q_SETDLIM
: Set disc quota limits and current usage for the user with ID *uid*. *Arg* is a major-minor device indicating a particular file system. *Addr* is a pointer to a struct dqblk structure (defined in <*sys/quota.h*>). This call is restricted to the super-user.

Q_GETDLIM
: Get disc quota limits and current usage for the user with ID *uid*. The remaining parameters are as for Q_SETDLIM.

Q_SETDUSE
: Set disc usage limits for the user with ID *uid*. *Arg* is a major-minor device indicating a particular file system. *Addr* is a pointer to a struct dqusage structure (defined in <*sys/quota.h*>). This call is restricted to the super-user.

Q_SYNC
: Update the on-disc copy of quota usages. The *uid*, *arg*, and *addr* parameters are ignored.

Q_SETUID
: Change the calling process's quota limits to those of the user with ID *uid*. The *arg* and *addr* parameters are ignored. This call is restricted to the super-user.

Q_SETWARN
: Alter the disc usage warning limits for the user with ID *uid*. *Arg* is a major-minor device indicating a particular file system. *Addr* is a pointer to a struct dqwarn structure (defined in <*sys/quota.h*>). This call is restricted to the super-user.

Q_DOWARN
: Warn the user with user ID *uid* about excessive disc usage. This call causes the system to check its current disc usage information and print a message on the terminal of the caller for each file system on which the user is over quota. If the *arg* parameter is specified as NODEV, all file systems which have disc quotas will be checked. Otherwise, *arg* indicates a specific major-minor device to be checked. This call is restricted to the super-user.

RETURN VALUE

A successful call returns 0 and, possibly, more information specific to the *cmd* performed; when an error occurs, the value -1 is returned and *errno* is set to indicate the reason.

ERRORS

A *quota* call will fail when one of the following occurs:

[EINVAL] *Cmd* is invalid.

[ESRCH] No disc quota is found for the indicated user.

[EPERM] The call is priviledged and the caller was not the super-user.

[EINVAL] The *arg* parameter is being interpreted as a major-minor device and it indicates an unmounted file system.

[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

setquota(2), quotaon(8), quotacheck(8)

BUGS

There should be someway to integrate this call with the resource limit interface provided by *setrlimit*(2) and *getrlimit*(2).

The Australian spelling of *disk* is used throughout the quota facilities in honor of the implementors.

NAME

read, readv – read input

SYNOPSIS

```
cc = read(d, buf, nbytes)
int cc, d;
char *buf;
int nbytes;

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

cc = readv(d, iov, iovcnt)
int cc, d;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

Read attempts to read *nbytes* of data from the object referenced by the descriptor *d* into the buffer pointed to by *buf*. *Readv* performs the same action, but scatters the input data into the *iovcnt* buffers specified by the members of the *iovec* array: iov[0], iov[1], ..., iov[iovcnt – 1].

For *readv*, 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 *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 a object is undefined.

Upon successful completion, *read* and *readv* return the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested if the descriptor references a file which has that many bytes left before the end-of-file, but in no other cases.

If the returned value is 0, then end-of-file has been reached.

RETURN VALUE

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

ERRORS

Read and *readv* will fail if one or more of the following are true:

[EBADF] *Fildes* is not a valid file descriptor open for reading.

[EFAULT] *Buf* points outside the allocated address space.

[EINTR] A read from a slow device was interrupted before any data arrived by the delivery of a signal.

In addition, *readv* may return one of the following errors:

[EINVAL] *Iovcnt* was 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.

SEE ALSO

dup(2), open(2), pipe(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:

[EPERM]	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), lstat(2), symlink(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" in the root file system of unit 0 of a disk chosen in a processor specific way. 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
crash(8S), halt(8), init(8), reboot(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 may be 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 send call is formed by *or*'ing one or more of the values,

```
#define MSG_PEEK    0x1    /* peek at incoming message */
#define MSG_OOB     0x2    /* process out-of-band data */
```

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). Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*.

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.

SEE ALSO

read(2), send(2), socket(2)

NAME
rename – change the name of a file

SYNOPSIS
rename(from, to)
char *from, *to;

DESCRIPTION
Rename causes the link named *from* to be renamed as *to*. If *to* exists, then it is first removed. 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.

CAVEAT
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", say "a/foo", being a hard link to directory "b", and an entry in directory "b", say "b/bar", 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.

RETURN VALUE
A 0 value is returned if the operation succeeds, otherwise *rename* returns -1 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:

[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.
[EPERM]	The file named by *from* is a directory and the effective user ID is not super-user.
[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]	*Path* points outside the process's allocated address space.
[EINVAL]	*From* is a parent directory of *to*.

SEE ALSO
open(2)

NAME

rmdir – remove a directory file

SYNOPSIS

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.
[EPERM]	The pathname contains a character 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.
[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.

SEE ALSO

mkdir(2), unlink(2)

NAME

select – synchronous I/O multiplexing

SYNOPSIS

#include <sys/time.h>

nfds = select(width, readfds, writefds, execptfds, timeout)
int width, *readfds, *writefds, *execptfds;
struct timeval *timeout;

DESCRIPTION

Select examines the I/O descriptors specified by the bit masks *readfds*, *writefds*, and *execptfds* to see if they are ready for reading, writing, or have an exceptional condition pending, respectively. *Width* is the number of significant bits in each bit mask that represent a file descriptor. Typically *width* has the value returned by *getdtablesize (2)* for the maximum number of file descriptors or is the constant 32 (number of bits in an int). File descriptor *f* is represented by the bit "1<<f" in the mask. *Select* returns, in place, a mask of those descriptors which are ready. The total number of ready descriptors is returned in *nfds*.

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 *execptfds* may be given as 0 if no descriptors are of interest.

RETURN VALUE

Select returns the number of descriptors which are contained in the bit masks, or –1 if an error occurred. If the time limit expires then *select* returns 0.

ERRORS

An error return from *select* indicates:

[EBADF] One of the bit masks specified an invalid descriptor.

[EINTR] An signal was delivered before any of the selected events occurred or the time limit expired.

SEE ALSO

accept(2), connect(2), gettimeofday(2), read(2), write(2), recv(2), send(2), getdtablesize(2)

BUGS

The descriptor masks are always modified on return, even if the call returns as the result of the timeout.

NAME

send, sendto, sendmsg – send a message from 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

S is a socket created with *socket*(2). *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 be set to SOF_OOB to send "out-of-band" data on sockets which support this notion (e.g. SOCK_STREAM).

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.

SEE ALSO

recv(2), socket(2)

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 NGRPS, as defined in *<sys/param.h>*.

Only the super-user may set new groups.

RETURN VALUE

A 0 value is returned on success, -1 on error, with a error code stored in *errno*.

ERRORS

The *setgroups* call will fail if:

[EPERM] The caller is not the super-user.

[EFAULT] The address specified for *gidset* is outside the process address space.

SEE ALSO

getgroups(2), initgroups(3)

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

setquota – enable/disable quotas on a file system

SYNOPSIS

setquota(special, file)
char *special, *file;

DESCRIPTION

Disc quotas are enabled or disabled with the *setquota* call. *Special* indicates a block special device on which a mounted file system exists. If *file* is nonzero, it specifies a file in that file system from which to take the quotas. If *file* is 0, then quotas are disabled on the file system. The quota file must exist; it is normally created with the *quotacheck*(8) program.

Only the super-user may turn quotas on or off.

SEE ALSO

quota(2), quotacheck(8), quotaon(8)

RETURN VALUE

A 0 return value indicates a successful call. A value of –1 is returned when an error occurs and *errno* is set to indicate the reason for failure.

ERRORS

Setquota will fail when one of the following occurs:

[NODEV]	The caller is not the super-user.
[NODEV]	*Special* does not exist.
[ENOTBLK]	*Special* is not a block device.
[ENXIO]	The major device number of *special* is out of range (this indicates no device driver exists for the associated hardware).
[EPERM]	The pathname contains a character with the high-order bit set.
[ENOTDIR]	A component of the path prefix in *file* is not a directory.
[EROFS]	*File* resides on a read-only file system.
[EACCES]	*File* resides on a file system different from *special*.
[EACCES]	*File* is not a plain file.

BUGS

The error codes are in a state of disarray; too many errors appear to the caller as one value.

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 super-user may change the real group ID of a process. Unpriviledged 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 super-user and a change other than changing the effective group-id to the real group-id was specified.

SEE ALSO

getgid(2), setreuid(2), setgid(3C)

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

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)

BUGS

The *how* values should be defined constants.

NAME

sigblock – block signals

SYNOPSIS

oldmask = sigblock(mask);
int mask;

DESCRIPTION

Sigblock adds the signals specified in *mask* to the set of signals currently being blocked from delivery. Signal *i* is blocked if the *i*–1'th bit in *mask* is a 1. The previous mask is returned, and may be restored using *sigsetmask*(2).

It is not possible to block SIGKILL, SIGSTOP, or SIGCONT; this restriction is silently imposed by the system.

RETURN VALUE

The previous set of masked signals is returned.

SEE ALSO

kill(2), sigvec(2), sigsetmask(2),

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. *Sigmask* is usually 0 to indicate that no signals are now to be blocked. *Sigpause* always terminates by being interrupted, returning EINTR.

In normal usage, a signal is blocked using *sigblock*(2), to begin a critical section, variables modified on the occurance 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)

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*–1'th bit in *mask* is a 1.

The system quietly disallows SIGKILL, SIGSTOP, or SIGCONT to be blocked.

RETURN VALUE

The previous set of masked signals is returned.

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_onstack;
};

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 occurence 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 initilized 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 *sv_onstack* is 1, 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 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 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 changed (see *win*(4S))

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 (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 *sv_handler* is SIG_IGN the signal is subsequently ignored, and pending instances of the signal are discarded.

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* or *write*(2) on a slow device (such as a terminal; but not a file) and during a *wait*(2).

After a *fork*(2) or *vfork*(2) the child inherits all signals, the signal mask, and the signal stack.

The *execve*(2) call resets all caught signals to default action; ignored signals remain ignored; the signal mask remains the same; the signal stack state is reset.

NOTES

The mask specified in *vec* is not allowed to block SIGKILL, SIGSTOP, or SIGCONT. This is done silently by the system.

RETURN VALUE

A 0 value indicated that the call succeeded. A −1 return value indicates an error occured 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), ptrace(2), kill(2), sigblock(2), sigsetmask(2), sigpause(2) sigstack(2), sigvec(2), setjmp(3), tty(4)

NOTES (VAX-11)

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 is either a constant as given below or, for compatibility mode faults, the code provided by the hardware (Compatibility mode faults are distinguished from the other SIGILL traps by having PSL_CM set in the psl). *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
Decimal overflow trap	SIGFPE	FPE_DECOVF_TRAP
Subscript-range	SIGFPE	FPE_SUBRNG_TRAP
Floating overflow fault	SIGFPE	FPE_FLTOVF_FAULT
Floating divide by zero fault	SIGFPE	FPE_FLTDIV_FAULT
Floating underflow fault	SIGFPE	FPE_FLTUND_FAULT
Length access control	SIGSEGV	
Protection violation	SIGBUS	
Reserved instruction	SIGILL	ILL_RESAD_FAULT
Customer-reserved instr.	SIGEMT	
Reserved operand	SIGILL	ILL_PRIVIN_FAULT
Reserved addressing	SIGILL	ILL_RESOP_FAULT
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	
Compatibility-mode	SIGILL	hardware supplied code
Chme	SIGSEGV	
Chms	SIGSEGV	
Chmu	SIGSEGV	

BUGS

This manual page is confusing.

NAME

socket – create an endpoint for communication

SYNOPSIS

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

s = socket(af, type, protocol)
int s, af, type, protocol;

DESCRIPTION

Socket creates an endpoint for communication and returns a descriptor.

The *af* parameter specifies an address format with which addresses specified in later operations using the socket should be interpreted. These formats are defined in the include file *<sys/socket.h>*. The currently understood formats are

AF_UNIX	(UNIX path names),
AF_INET	(ARPA Internet addresses),
AF_PUP	(Xerox PUP-I Internet addresses), and
AF_IMPLINK	(IMP "host at IMP" addresses).

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 with an out-of-band data transmission mechanism. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). SOCK_RAW sockets provide access to internal network interfaces. The types SOCK_RAW, which is available only to the super-user, and SOCK_SEQPACKET and SOCK_RDM, which are 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 using a given address format. 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 *services*(3N) and *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_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in *send*(2) calls. It is also possible to receive datagrams at such a socket with *recv*(2).

An *fcntl*(2) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives.

The operation of sockets is controlled by socket level *options*. These options are defined in the file <*sys/socket.h*> and explained below. *Setsockopt* and *getsockopt*(2) are used to set and get options, respectively.

SO_DEBUG	turn on recording of debugging information
SO_REUSEADDR	allow local address reuse
SO_KEEPALIVE	keep connections alive
SO_DONTROUTE	do no apply routing on outgoing messages
SO_LINGER	linger on close if data present
SO_DONTLINGER	do not linger on close

SO_DEBUG enables debugging in the underlying protocol modules. SO_REUSEADDR indicates 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 and SO_DONTLINGER control the actions taken when unsent messags 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_DONTLINGER is specified and a *close* is issued, the system will process the close in a manner which allows the process to continue as quickly as possible.

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:

[EAFNOSUPPORT] The specified address family is not supported in this version of the system.

[ESOCKTNOSUPPORT] The specified socket type is not supported in this address family.

[EPROTONOSUPPORT] The specified protocol is not supported.

[EMFILE] The per-process descriptor table is full.

[ENOBUFS] No buffer space is available. The socket cannot be created.

SEE ALSO

accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), recv(2), select(2), send(2), shutdown(2), socketpair(2)
"A 4.2BSD Interprocess Communication Primer".

BUGS

The use of keepalives is a questionable feature for this layer.

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* system 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.
[EOPNOSUPPORT]	The specified protocol does not support creation of socket pairs.
[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, lstat, fstat – get file status

SYNOPSIS

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

stat(path, buf)
char *path;
struct stat *buf;

lstat(path, buf)
char *path;
struct stat *buf;

fstat(fd, buf)
int fd;
struct stat *buf;
```

DESCRIPTION

Stat obtains information about the file *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 reachable.

Lstat is like *stat* 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.

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*

```
struct stat {
        dev_t     st_dev;       /* device inode resides on */
        ino_t     st_ino;       /* this inode's number */
        u_short   st_mode;      /* protection */
        short     st_nlink;     /* number or hard links to the file */
        short     st_uid;       /* user-id of owner */
        short     st_gid;       /* group-id of owner */
        dev_t     st_rdev;      /* the device type, for inode that is device */
        off_t     st_size;      /* total size of file */
        time_t    st_atime;     /* file last access time */
        int       st_spare1;
        time_t    st_mtime;     /* file last modify time */
        int       st_spare2;
        time_t    st_ctime;     /* file last status change time */
        int       st_spare3;
        long      st_blksize;   /* optimal blocksize for file system i/o ops */
        long      st_blocks;    /* actual number of blocks allocated */
        long      st_spare4[2];
};
```

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, although 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), *utimes*(2), *write*(2).

st_ctime Time when file status was last changed. It is set both both by writing and changing the i-node. Changed by the following system calls: *chmod*(2) *chown*(2), *link*(2), *mknod*(2), *unlink*(2), *utimes*(2), *write*(2), *truncate*(2).

The status information word *st_mode* has bits:

```
#define S_IFMT      0170000   /* type of file */
#define   S_IFDIR   0040000   /* directory */
#define   S_IFCHR   0020000   /* character special */
#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   /* save swapped text even after use */
#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)).

When *fd* is associated with a pipe, *fstat* reports an ordinary file with an i-node number, restricted permissions, and a not necessarily meaningful length.

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 and *lstat* will fail if one or more of the following are true:

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The pathname contains a character with the high-order bit set.

[ENOENT] The pathname was too long.

[ENOENT] The named file does not exist.

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

[EFAULT] *Buf* or *name* points to an invalid address.

Fstat will fail if one or both of the following are true:

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

[EFAULT] *Buf* points to an invalid address.

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

CAVEAT

The fields in the stat structure currently marked *st_spare1*, *st_spare2*, and *st_spare3* are present in preparation for inode time stamps expanding to 64 bits. This, however, can break certain programs which depend on the time stamps being contiguous (in calls to *utimes*(2)).

SEE ALSO

chmod(2), chown(2), utimes(2)

BUGS

Applying *fstat* to a socket returns a zero'd buffer.

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. The size of the swap area on *special* is calculated at the time the device is first made available for swapping.

SEE ALSO

swapon(8), config(8)

BUGS

There is no way to stop swapping on a disk so that the pack may be dismounted.

This call will be upgraded in future versions of the system.

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:

[EPERM]	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), cron(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 register d0 value of the system call is returned.

DIAGNOSTICS

When the C-bit is set, *syscall* returns –1 and sets the external variable *errno* (see *intro*(2)).

BUGS

There is no way to simulate system calls such as *pipe*(2), which return values in register d1.

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:

[EPERM]	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

oumask = umask(numask)
int oumask, numask;

DESCRIPTION

Umask sets the process's file mode creation mask to *numask* and returns the previous value of the mask. The low-order 9 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 022 (write access for 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)

NAME

unlink – remove directory entry

SYNOPSIS

unlink(path)
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. If, however, the file was open in any process, the actual resource reclamation is delayed until it is closed, even though the directory entry has disappeared.

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

The *unlink* succeeds unless:

[EPERM]	The path contains a character with the high-order bit set.
[ENOENT]	The path name is too long.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[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.

SEE ALSO

close(2), link(2), rmdir(2)

NAME

utimes – set file times

SYNOPSIS

```
#include <sys/types.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*.

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

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

[EPERM]	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 and *times* is NULL and write access is denied.
[EROFS]	The file system containing the file 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 **<vadvise.h>**. Currently, two calls t *vadvise* are implemented.

The call

vadvise(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, and more emphasis on referenceless circular behavior. 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.

The call

vadvise(VA_NORM);

restores default paging replacement behavior after a call to

vadvise(VA_ANOM);

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 *ioctls* 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 access the terminal by the affected processes will yield i/o errors (EBADF). Finally, a hangup signal (SIGHUP) is sent to the process group of the control terminal.**

SEE ALSO

init (8)

BUGS

Access to the control terminal via /dev/tty is still possible.

This call should be replaced by an automatic mechanism which takes place on process exit.

NAME

wait, wait3 – wait for process to terminate or stop

SYNOPSIS

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

pid = wait(status)
int pid;
union wait *status;

pid = wait(0)
int pid;

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

pid = wait3(status, options, rusage)
int pid;
union wait *status;
int options;
struct rusage *rusage;
```

DESCRIPTION

Wait causes its caller to delay until a signal is received or one of its child processes terminates or stops due to tracing. If any child has died or stopped due to tracing and this has not been reported via *wait*, return is immediate, returning the process id and exit status of one of those children. If that child had died, it is discarded. If there are no children, return is immediate with the value –1 returned. If there are only running or stopped but reported children, the calling processes is suspended.

On return from a successful *wait* call, *status* is nonzero, and the high byte of *status* contains the low byte of the argument to *exit* supplied by the child process; the low byte of *status* contains the termination status of the process. A more precise definition of the *status* word is given in *<sys/wait.h>*.

Wait3 is an alternate interface which allows both non-blocking status collection and the status of children stopped by any means. The *status* parameter is defined as above. The *options* parameter is used to indicate the call should not block if there are no processes which have status to report (WNOHANG), and/or that children of the current process which are stopped due to a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal are eligible to have their status reported as well (WUNTRACED). A terminated child is discarded after it reports status, and a stopped process will not report its status more than once. 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.)

When the WNOHANG option is specified and no processes have status to report, *wait3* returns a *pid* of 0. The WNOHANG and WUNTRACED options may be combined by *or*'ing the two values.

NOTES

See *sigvec*(2) for a list of termination statuses (signals); 0 status indicates normal termination. A special status (0177) is returned for a stopped process which has not terminated and can be restarted; see *ptrace*(2) and *sigvec*(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.

Wait and *wait3* are automatically restarted when a process receives a signal while awaiting termination of a child process.

RETURN VALUE

If *wait* returns due to a stopped due to tracing or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

Wait3 returns -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 are true:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EFAULT] The *status* or *rusage* arguments point to an illegal address.

SEE ALSO

exit(2)

NAME

write, writev – write on a file

SYNOPSIS

```
write(d, buf, nbytes)
int d;
char *buf;
int nbytes;

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

writev(d, iov, ioveclen)
int d;
struct iovec *iov;
int ioveclen;
```

DESCRIPTION

Write attempts to write *nbytes* of data to the object referenced by the descriptor *d* from the buffer pointed to by *buf*. *Writev* performs the same action, but gathers the output data from the *iovlen* buffers specified by the members of the *iov* array: iov[0], iov[1], etc.

On objects capable of seeking, the *write* starts at a position given by the pointer associated with *d*, see *lseek*(2). Upon return from *write*, the pointer is incremented by the number of bytes actually written.

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 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.

RETURN VALUE

Upon successful completion the number of bytes actually writen is returned. Otherwise 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:

[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.
[EPIPE]	An attempt is made to write to a socket of type SOCK_STREAM which is not connected to a peer socket.
[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.

SEE ALSO

lseek(2), open(2), pipe(2)

NAME

intro – introduction to library functions

DESCRIPTION

Section 3 describes functions found in libraries.

This section describes subroutines found in the system libraries.

The main C library is /lib/libc.a, and contains all the system call entry points described in section 2 as well as functions described in several subsections here. The primary functions in the C library are described in the main section 3. Functions associated with the "standard I/O library" used by many C programs are found in section 3S. The libc library also includes the Internet network functions described in section 3N and routines providing compatibility with other UNIX systems as described in section 3C as well as all the system entry points from section 2.

Other sections here are:

(3F) The 3F functions are all functions callable from FORTRAN. These functions perform the same jobs as the straight "3" functions do for C programmers. There are in fact three FORTRAN libraries, namely **–lU77** which contains the system interface routines, **–lI77** which is the I/O interface library, and **–lF77** which is everything not contained in the other two. These libraries are searched automatically by the loader when loading FORTRAN programs.

(3M) These functions constitute the math library. C declarations for the types of functions may be obtained from the include file <*math.h*>. To use these functions with C programs use a **–lm** option with *cc*(1). They are automatically loaded as needed by the Fortran and Pascal compilers *f77*(1) and *pc*(1).

(3X) Various specialized libraries have not been given distinctive captions. Files in which such libraries are found are named on appropriate pages if they don't appear in the *libc* library.

FILES

/lib/libc.a	C Library ((2), (3), (3N) and (3C) routines)
/usr/lib/libc_p.a	Profiling C library (for *gprof*(1))
/usr/lib/libm.a	Math Library **–lm** (see section 3M)
/usr/lib/libm_p.a	Profiling version of **–lm**
/usr/lib/libU77.a	FORTRAN system interface (see section 3F)
/usr/lib/libI77.a	FORTRAN I/O (see section 3F)
/usr/lib/libF77.a	FORTRAN everything else (see section 3F)
/usr/lib/libcurses.a	screen management routines (see *curses*(3X)
/usr/lib/libdbm.a	data base management routines (see *dbm*(3X))
/usr/lib/libmp.a	multiple precision math library (see *mp*(3X))
/usr/lib/libtermcap.a	terminal handling routines (see *termcap*(3X))
/usr/lib/libtermcap_p.a	
/usr/lib/libtermlib	
/usr/lib/libtermlib_p.a	
/usr/lib/libplot.a	plot routines (see *plot*(3X))
/usr/lib/lib300.a	"
/usr/lib/lib300s.a	"
/usr/lib/lib4014.a	"
/usr/lib/lib450.a	"

SEE ALSO

intro(3C), intro(3S), intro(3F), intro(3M), intro(3N), nm(1), ld(1), cc(1), f77(1), intro(2)

DIAGNOSTICS

Functions in the math library (section 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)) 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*>.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
abort	abort.3	generate a fault
abs	abs.3	integer absolute value
alarm	alarm.3c	schedule signal after specified time
alloca	malloc.3	memory allocator
alphasort	scandir.3	scan a directory
asctime	ctime.3	convert date and time to ASCII
assert	assert.3	program verification
atof	atof.3	convert ASCII to numbers
atoi	atof.3	convert ASCII to numbers
atol	atof.3	convert ASCII to numbers
bcmp	bstring.3	bit and byte string operations
bcopy	bstring.3	bit and byte string operations
bzero	bstring.3	bit and byte string operations
calloc	malloc.3	memory allocator
cfree	malloc.3	memory allocator
clearerr	ferror.3s	stream status inquiries
closedir	directory.3	directory operations
closelog	syslog.3	control system log
crypt	crypt.3	DES encryption
ctime	ctime.3	convert date and time to ASCII
dysize	ctime.3	convert date and time to ASCII
ecvt	ecvt.3	output conversion
edata	end.3	last locations in program
encrypt	crypt.3	DES encryption
end	end.3	last locations in program
endfsent	getfsent.3	get file system descriptor file entry
endgrent	getgrent.3	get group file entry
endhostent	gethostent.3n	get network host entry
endnetent	getnetent.3n	get network entry
endprotoent	getprotoent.3n	get protocol entry
endpwent	getpwent.3	get password file entry
endservent	getservent.3n	get service entry
environ	execl.3	execute a file
errno	perror.3	system error messages
etext	end.3	last locations in program
execl	execl.3	execute a file
execle	execl.3	execute a file
execlp	execl.3	execute a file
execv	execl.3	execute a file
execvp	execl.3	execute a file
exit	exit.3	terminate a process after flushing any pending output
fclose	fclose.3s	close or flush a stream
fcvt	ecvt.3	output conversion
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
ffs	bstring.3	bit and byte string operations
fgetc	getc.3s	get character or integer 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
free	malloc.3	memory allocator
freopen	fopen.3s	open a stream
frexp	frexp.3	split into mantissa and exponent
fscanf	scanf.3s	formatted input conversion
fseek	fseek.3s	reposition a stream
ftell	fseek.3s	reposition a stream
ftime	time.3c	get date and time
fwrite	fread.3s	buffered binary input/output
gcvt	ecvt.3	output conversion
getc	getc.3s	get character or integer from stream
getchar	getc.3s	get character or integer from stream
getenv	getenv.3	value for environment name
getfsent	getfsent.3	get file system descriptor file entry
getfsfile	getfsent.3	get file system descriptor file entry
getfsspec	getfsent.3	get file system descriptor file entry
getfstype	getfsent.3	get file system descriptor file entry
getgrent	getgrent.3	get group file entry
getgrgid	getgrent.3	get group file entry
getgrnam	getgrent.3	get group file entry
gethostbyaddr	gethostent.3n	get network host entry
gethostbyname	gethostent.3n	get network host entry
gethostent	gethostent.3n	get network host entry
getlogin	getlogin.3	get login name
getnetbyaddr	getnetent.3n	get network entry
getnetbyname	getnetent.3n	get network entry
getnetent	getnetent.3n	get network entry
getopt	getopt.3c	get option letter from argv
getpass	getpass.3	read a password
getprotobyname	getprotoent.3n	get protocol entry
getprotobynumber	getprotoent.3n	get protocol entry
getprotoent	getprotoent.3n	get protocol entry
getpw	getpw.3	get name from uid
getpwent	getpwent.3	get password file entry
getpwnam	getpwent.3	get password file entry
getpwuid	getpwent.3	get password file entry
gets	gets.3s	get a string from a stream
getservbyname	getservent.3n	get service entry
getservbyport	getservent.3n	get service entry
getservent	getservent.3n	get service entry
getw	getc.3s	get character or integer from stream
getwd	getwd.3	get current working directory pathname
gmtime	ctime.3	convert date and time to ASCII
gtty	stty.3c	set and get terminal state
htonl	byteorder.3n	convert values between host and network byte order
htons	byteorder.3n	convert values between host and network byte order
index	string.3	string operations
inet_addr	inet.3n	Internet address manipulation

inet_lnaof	inet.3n	Internet address manipulation
inet_makeaddr	inet.3n	Internet address manipulation
inet_netof	inet.3n	Internet address manipulation
inet_network	inet.3n	Internet address manipulation
inet_ntoa	inet.3n	Internet address manipulation
initgroups	initgroups.3	initialize group access list
initstate	random.3	better random number generator; routines for changing generators
insque	insque.3	insert/remove element from a queue
isalnum	ctype.3	character classification macros
isalpha	ctype.3	character classification macros
isascii	ctype.3	character classification macros
isatty	ttyname.3	find name of a terminal
iscntrl	ctype.3	character classification macros
isdigit	ctype.3	character classification macros
isgraph	ctype.3	character classification macros
isinf	isinf.3	test for indeterminate floating point values
islower	ctype.3	character classification macros
isnan	isinf.3	test for indeterminate floating point values
isprint	ctype.3	character classification macros
ispunct	ctype.3	character classification macros
isspace	ctype.3	character classification macros
isupper	ctype.3	character classification macros
ldexp	frexp.3	split into mantissa and exponent
localtime	ctime.3	convert date and time to ASCII
longjmp	setjmp.3	non-local goto
malloc	malloc.3	memory allocator
mktemp	mktemp.3	make a unique file name
modf	frexp.3	split into mantissa and exponent
moncontrol	monitor.3	prepare execution profile
monitor	monitor.3	prepare execution profile
monstartup	monitor.3	prepare execution profile
nice	nice.3c	set program priority
nlist	nlist.3	get entries from name list
ntohl	byteorder.3n	convert values between host and network byte order
ntohs	byteorder.3n	convert values between host and network byte order
opendir	directory.3	directory operations
openlog	syslog.3	control system log
optarg	getopt.3c	get option letter from argv
optind	getopt.3c	get option letter from argv
pause	pause.3c	stop until signal
pclose	popen.3s	initiate I/O to/from a process
perror	perror.3	system error messages
popen	popen.3s	initiate I/O to/from a process
printf	printf.3s	formatted output conversion
psignal	psignal.3	system signal messages
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
qsort	qsort.3	quicker sort
rand	rand.3c	random number generator
random	random.3	better random number generator; routines for changing generators
rcmd	rcmd.3n	routines for returning a stream to a remote command

re_comp	regex.3	regular expression handler
re_exec	regex.3	regular expression handler
readdir	directory.3	directory operations
realloc	malloc.3	memory allocator
remque	insque.3	insert/remove element from a queue
rewind	fseek.3s	reposition a stream
rewinddir	directory.3	directory operations
rexec	rexec.3n	return stream to a remote command
rindex	string.3	string operations
rresvport	rcmd.3n	routines for returning a stream to a remote command
ruserok	rcmd.3n	routines for returning a stream to a remote command
scandir	scandir.3	scan a directory
scanf	scanf.3s	formatted input conversion
seekdir	directory.3	directory operations
setbuf	setbuf.3s	assign buffering to a stream
setbuffer	setbuf.3s	assign buffering to a stream
setegid	setuid.3	set user and group ID
seteuid	setuid.3	set user and group ID
setfsent	getfsent.3	get file system descriptor file entry
setgid	setuid.3	set user and group ID
setgrent	getgrent.3	get group file entry
sethostent	gethostent.3n	get network host entry
setjmp	setjmp.3	non-local goto
setkey	crypt.3	DES encryption
setlinebuf	setbuf.3s	assign buffering to a stream
setnetent	getnetent.3n	get network entry
setprotoent	getprotoent.3n	get protocol entry
setpwent	getpwent.3	get password file entry
setrgid	setuid.3	set user and group ID
setruid	setuid.3	set user and group ID
setservent	getservent.3n	get service entry
setstate	random.3	better random number generator; routines for changing generators
setuid	setuid.3	set user and group ID
signal	signal.3	simplified software signal facilities
sleep	sleep.3	suspend execution for interval
sprintf	printf.3s	formatted output conversion
srand	rand.3c	random number generator
srandom	random.3	better random number generator; routines for changing generators
sscanf	scanf.3s	formatted input conversion
stdio	intro.3s	standard buffered input/output package
strcat	string.3	string operations
strcmp	string.3	string operations
strcpy	string.3	string operations
strlen	string.3	string operations
strncat	string.3	string operations
strncmp	string.3	string operations
strncpy	string.3	string operations
stty	stty.3c	set and get terminal state
swab	swab.3	swap bytes
sys_errlist	perror.3	system error messages
sys_nerr	perror.3	system error messages
sys_siglist	psignal.3	system signal messages
syslog	syslog.3	control system log

system	system.3	issue a shell command
telldir	directory.3	directory operations
time	time.3c	get date and time
times	times.3c	get process times
timezone	ctime.3	convert date and time to ASCII
tmpnam	tmpnam.3c	create a name for a temporary file
ttyname	ttyname.3	find name of a terminal
ttyslot	ttyname.3	find name of a terminal
ulimit	ulimit.3c	get and set user limits
ungetc	ungetc.3s	push character back into input stream
utime	utime.3c	set file times
valloc	valloc.3	aligned memory allocator
varargs	varargs.3	variable argument list
vlimit	vlimit.3c	control maximum system resource consumption
vtimes	vtimes.3c	get information about resource utilization

NAME

abort – generate a fault

DESCRIPTION

Abort executes an instruction which is illegal in user mode. This causes a signal that normally terminates the process with a core dump, which may be used for debugging.

SEE ALSO

adb(1S), signal(3), exit(2)

DIAGNOSTICS

Usually 'IOT trap – core dumped' from the shell.

BUGS

The *abort* function does not flush standard I/O buffers. Use *fflush* as described in *fclose*(3S).

NAME

abs – integer absolute value

SYNOPSIS

abs(i)
int i;

DESCRIPTION

Abs returns the absolute value of its integer operand.

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

assert – program verification

SYNOPSIS

#include <assert.h>

assert(expression)

DESCRIPTION

Assert is a macro that indicates *expression* is expected to be true at this point in the program. It causes an *exit*(2) with a diagnostic comment on the standard output when *expression* is false (0). Compiling with the *cc*(1) option –DNDEBUG effectively deletes *assert* from the program.

DIAGNOSTICS

'Assertion failed: file *f* line *n*.' *F* is the source file and *n* the source line number of the *assert* statement.

NAME

atof, atoi, atol – convert ASCII to numbers

SYNOPSIS

double atof(nptr)
char *nptr;

atoi(nptr)
char *nptr;

long atol(nptr)
char *nptr;

DESCRIPTION

These functions convert a string pointed to by *nptr* to floating, integer, and long integer representation 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 and *atol* recognize an optional string of spaces, then an optional sign, then a string of digits.

SEE ALSO

scanf(3S)

BUGS

There are no provisions for overflow.

Currently, *atof* performs highly inaccurate conversions of very large or very small numbers — on the order of 10**32 or its reciprocal.

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 *string*(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 *b*.

Ffs finds the first bit set in the argument passed it and returns the index of that bit. Bits are numbered starting at 1 from the right. A return value of –1 indicates the value passed is zero.

BUGS

The *bcmp* and *bcopy* routines take parameters backwards from *strcmp* and *strcpy*.

NAME

crypt, setkey, encrypt – DES encryption

SYNOPSIS

```
char *crypt(key, salt)
char *key, *salt;

setkey(key)
char *key;

encrypt(block, edflag)
char *block;
```

DESCRIPTION

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 non-zero, it is decrypted.

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

ctime, localtime, gmtime, asctime, timezone, dysize – convert date and time to ASCII

SYNOPSIS

char *ctime(clock)
long *clock;

#include <sys/time.h>

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;

char *timezone(zone, dst)

int dysize(y)
int y;

DESCRIPTION

Ctime converts a time pointed to by *clock* such as returned by *gettimeofday*(2) 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;
        int     tm_min;
        int     tm_hour;
        int     tm_mday;
        int     tm_mon;
        int     tm_year;
        int     tm_wday;
        int     tm_yday;
        int     tm_isdst;
};
```

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.

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.

Dysize returns the number of days in the argument year, either 365 or 366.

SEE ALSO

gettimeofday(2), time(3C)

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, iscntrl, isascii, isgraph, toupper, tolower, toascii – character classification and conversion macros

SYNOPSIS

#include <ctype.h>

isalpha(c)

. . .

CHARACTER CLASSIFICATION MACROS

These 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(*c*) is true and on the single non-ASCII value EOF (see *stdio*(3S)).

isalpha(*c*)	*c* is a letter
isupper(*c*)	*c* is an upper case letter
islower(*c*)	*c* is a lower case letter
isdigit(*c*)	*c* is a digit
isxdigit(*c*)	*c* is a hexadecimal digit
isalnum(*c*)	*c* is an alphanumeric character
isspace(*c*)	*c* is a space, tab, carriage return, newline, or formfeed
ispunct(*c*)	*c* is a punctuation character (neither control nor alphanumeric)
isprint(*c*)	*c* is a printing character, code 040(8) (space) through 0176 (tilde)
iscntrl(*c*)	*c* is a delete character (0177) or ordinary control character (less than 040).
isascii(*c*)	*c* is an ASCII character, code less than 0200
isgraph(*c*)	*c* is a visible graphic character, code 041 (exclamation mark) through 0176 (tilde).

CHARACTER CONVERSION MACROS

These macros perform simple conversions on single characters.

toupper(*c*)	converts *c* to its upper-case equivalent. Note that this *only* works where *c* is known to be a lower-case character to start with (presumably checked via *islower*).
tolower(*c*)	converts *c* to its lower-case equivalent. Note that this *only* works where *c* is known to be a upper-case character to start with (presumably checked via *isupper*).
toascii(*c*)	masks *c* with the correct value so that *c* is guaranteed to be an ASCII character in the range 0 thru 0x7f.

SEE ALSO

ascii(7)

NAME

opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

SYNOPSIS

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

DIR *opendir(filename)
char *filename;

struct direct *readdir(dirp)
DIR *dirp;

long telldir(dirp)
DIR *dirp;

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

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 is not a directory, 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.

Telldir returns the current location associated with the named *directory stream*.

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*.

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 which searchs 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;
```

SEE ALSO

open(2), close(2), read(2), lseek(2), dir(5)

BUGS

Old UNIX programs which examine directories should be converted to use this package, as the new directory format is non-obvious.

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

isinf(3), 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*(3S)), 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

execl, execv, execle, execlp, execvp, environ – execute a file

SYNOPSIS

execl(name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ..., *argn;

execv(name, argv)
char *name, *argv[ ];

execle(name, arg0, arg1, ..., argn, 0, envp)
char *name, *arg0, *arg1, ..., *argn, *envp[ ];

execlp(name, arg0, arg1, ..., argn, 0, envp)
char *name, *arg0, *arg1, ..., *argn, *envp[ ];

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

extern char **environ;

DESCRIPTION

These routines provide various interfaces to the *execve* 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 0 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 0 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 one 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*(5) 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.

FILES

/bin/sh shell, invoked if command file found by *execlp* or *execvp*

SEE ALSO

execve(2), fork(2), environ(5), csh(1), sh(1)
"UNIX Programming" in *Programming Tools for the SUN Workstation*, pp. 1-3.

DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not start with a valid magic number (see *a.out*(5)), if maximum memory is exceeded, or if the arguments require too much space, a return constitutes the diagnostic; the return value is –1. Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed.

BUGS

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.

NAME

exit – terminate a process after flushing any pending output

SYNOPSIS

exit(status)
int status;

DESCRIPTION

Exit terminates a process by calling *exit*(2) after calling the Standard I/O library function *_cleanup* to flush any buffered output. *Exit* never returns.

SEE ALSO

exit(2), intro(3S)

NAME

frexp, ldexp, modf – split into mantissa and exponent

SYNOPSIS

double frexp(value, eptr)
double value;
int *eptr;

double ldexp(value, exp)
double value;

double modf(value, iptr)
double value, *iptr;

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*.

SEE ALSO

isinf(3)

BUGS

The identity claimed for the results of *frexp* cannot hold when the *value* argument is an IEEE indefinite quantity — infinity or not-a-number.

NAME

getdate – convert time and date from ASCII

SYNOPSIS

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

time_t getdate(buf, now)
char *buf;
struct timeb *now;

DESCRIPTION

Getdate converts most common time specifications to standard UNIX format. The first argument is the character string containing the time and date; the second is the assumed current time (used for relative specifications); if **NULL** is passed, *ftime*(2) is used to obtain the current time and timezone.

The character string consists of 0 or more specifications of the following form:

tod
: A *tod* is a time of day, which is of the form *hh*:*mm*[:*ss*] (or *hhmm*) [*meridian*] [*zone*]. If no meridian – **am** or **pm** – is specified, a 24-hour clock is used. A *tod* may be specified as just *hh* followed by a *meridian*.

date
: A *date* is a specific month and day, and possibly a year. Acceptable formats are *mm*/*dd*[/*yy*] and *monthname dd*[, *yy*] If omitted, the year defaults to the current year; if a year is specified as a number less than 100, 1900 is added. If a number not followed by a day or relative time unit occurs, it will be interpreted as a year if a *tod*, *monthname*, and *dd* have already been specified; otherwise, it will be treated as a *tod*. This rule allows the output from *date*(1) or *ctime*(3) to be passed as input to *getdate*.

day
: A *day* of the week may be specified; the current day will be used if appropriate. A *day* may be preceeded by a *number*, indicating which instance of that day is desired; the default is **1**. Negative *numbers* indicate times past. Some symbolic *numbers* are accepted: **last**, **next**, and the ordinals **first** through **twelfth** (**second** is ambiguous, and is not accepted as an ordinal number). The symbolic number **next** is equivalent to **2**; thus, *next monday* refers not to the immediately coming Monday, but to the one a week later.

relative time
: Specifications relative to the current time are also accepted. The format is [*number*] *unit*; acceptable units are **year**, **month**, **fortnight**, **week**, **day**, **hour**, **minute**, and **second**.

The actual date is formed as follows: first, any absolute date and/or time is processed and converted. Using that time as the base, day-of-week specifications are added; last, relative specifications are used. If a date or day is specified, and no absolute or relative time is given, midnight is used. Finally, a correction is applied so that the correct hour of the day is produced after allowing for daylight savings time differences.

Getdate accepts most common abbreviations for days, months, etc.; in particular, it recognizes them with upper or lower case first letter, and recognizes three-letter abbreviations for any of them, with or without a trailing period. Units, such as **weeks**, may be specified in the singular or plural. Timezone and meridian values may be in upper or lower case, and with or without periods.

FILES

/usr/lib/libu.a

SEE ALSO

ctime(3), time(2)

BUGS

Because *yacc*(1) is used to parse the date, *getdate* cannot be used a subroutine to any program that also needs *yacc*.

The grammar and scanner are rather primitive; certain desirable and unambiguous constructions are not accepted. Worse yet, the meaning of some legal phrases is not what is expected; *next week* is identical to *2 weeks*.

The daylight savings time correction is not perfect, and can get confused if handed times between midnight and 2:00 am on the days that the reckoning changes.

Because *localtime*(2) accepts an old-style time format without zone information, attempting to pass *getdate* a current time containing a different zone will probably fail.

NAME

getenv – value for environment name

SYNOPSIS

```
char *getenv(name)
char *name;
```

DESCRIPTION

Getenv searches the environment list (see *environ*(5)) for a string of the form *name*=*value* and returns a pointer to the string *value* if such a string is present, otherwise *getenv* returns the value 0 (NULL).

SEE ALSO

environ(5), execve(2)

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

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.

FILES

/etc/fstab

SEE ALSO

fstab(5)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

The return value points to static information which is overwritten in each call.

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent – get group file entry

SYNOPSIS

```
#include <grp.h>

struct group *getgrent()

struct group *getgrgid(gid)
int gid;

struct group *getgrnam(name)
char *name;

setgrent()

endgrent()
```

DESCRIPTION

Getgrent, getgrgid and *getgrnam* each return pointers to an object with the following structure containing the broken-out fields of a line in the group 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	Null-terminated vector of pointers to the individual member names.

Getgrent simply reads the next line while *getgrgid* and *getgrnam* search until a matching *gid* 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.

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.

FILES

/etc/group

SEE ALSO

getlogin(3), getpwent(3), group(5)

DIAGNOSTICS

A null pointer (0) is returned on EOF or error.

BUGS

The return value points to static information which is overwritten on each call.

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 *getpwnam* 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 typewriter, 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.

Getlogin does not work for processes running under a *pty* (for example, emacs shell buffers, or shell tools) unless the program "fakes" the login name in the */etc/utmp* file.

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

getpw – get name from uid

SYNOPSIS

getpw(uid, buf)
char *buf;

DESCRIPTION

Getpw is obsoleted by getpwent(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

getwd – get current working directory pathname

SYNOPSIS

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

char *getwd(pathname)
char pathname[MAXPATHLEN];
```

DESCRIPTION

Getwd **copies the absolute pathname of the current working directory to** ***pathname*** **and returns a pointer to the result.**

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

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*.

Noone 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 be in the form of "struct qelem". *Insque* inserts *elem* in a queue imediately after *pred*; *remque* removes an entry *elem* from a queue.

SEE ALSO

"VAX Architecture Handbook", pp. 228-235. It does work on SUNS.

NAME

isinf, isnan – test for indeterminate floating point values

SYNOPSIS

int isinf(value)
double value;

int isnan(value)
double value;

DESCRIPTION

Isinf returns a value of 1 if its *value* is an IEEE format infinity (two words 0x7ff00000 0x00000000) or an IEEE negative infinity, and returns a zero otherwise.

Isnan returns a value of 1 if its *value* is an IEEE format 'not-a-number' (two words 0x7ff *nnnnn* 0x *nnnnnnnn*) where *n* is not zero) or its negative, and returns a zero otherwise.

Some library routines such as *ecvt*(3) do not handle indeterminate floating point values gracefully. Prospective arguments to such routines should be checked with *isinf* or *isnan* before calling these routines.

BUGS

Need a manual section describing the format of IEEE numbers in detail.

NAME

malloc, free, realloc, calloc, cfree, alloca – memory allocator

SYNOPSIS

char *malloc(size)
unsigned size;

free(ptr)
char *ptr;

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

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

cfree(ptr)
char *ptr;

char *alloca(size)
int size;

malloc_ok(size)
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 a cartesian tree of free blocks. 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.

Realloc also works if *ptr* points to a block freed since the last call of *malloc, realloc* or *calloc*.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros, and can be freed with *cfree*.

Alloca allocates *size* bytes of space in the stack frame of the caller. This temporary space is automatically freed on return.

Ocassionally a program will overun the storage allocated from *Malloc* . *Malloc_ok* helps determine when this has happened. It checks all blocks (free or allocated) looking for duplicates, strange addresses and absurd sizes. *Malloc_ok returns true if everything is* is found. The *size* parameter specifies the maximum acceptable size of a block. A block with a larger size is considered bad. If *size* is zero a maximum of 10000 is assumed.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

SEE ALSO

Fast Fits by C. J. Stephenson

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.

BUGS

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

Alloca is machine dependent; it's use is discouraged.

NAME

mktemp – make a unique file name

SYNOPSIS

char *mktemp(template)
char *template;

DESCRIPTION

Mktemp replaces *template* by a unique file name, and returns the address of the template. The template should look like a file name with six trailing X's, which will be replaced with the current process id and a unique letter.

Notes:

- *Mktemp* actually *changes* the template string which you pass, this means that you cannot use the same template string more than once — you need a fresh template for every unique file you want to open.
- When *mktemp* is creating a new unique filename it checks for the prior existence of a file with that name. This means that if you are creating more than one unique filename, it is bad practice to use the same root template for multiple invocations of *mktemp*.

SEE ALSO

getpid(2)

NAME

monitor, monstartup, moncontrol – prepare execution profile

SYNOPSIS

monitor(lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)(), (*highpc)();
short 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 profil buffer allocation. An executable program created by:

cc –pg . . .

automatically includes calls for the *gprof*(1) monitor.

Monstartup is a high level interface to *profil*(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(0x8000, 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 irregardless of the state of *moncontrol.*

Monitor is a low level interface to *profil*(2). *Lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* short integers. 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 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(0x8000, etext, buf, bufsize, nfunc);
```

FILES

mon.out

SEE ALSO

cc(1), prof(1), gprof(1), profil(2), sbrk(2)

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

perror, sys_errlist, sys_nerr, errno – system error messages

SYNOPSIS

perror(s)
char *s;

int sys_nerr;
char *sys_errlist[];

int errno;

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. Most usefully, the argument string is 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.

To simplify variant formatting of messages, the vector of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the newline. *Sys_nerr* is the number of messages provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2), psignal(3)

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

perror(3), signal(3)

NAME

qsort – quicker sort

SYNOPSIS

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

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.

SEE ALSO

sort(1)

NAME

random, srandom, initstate, setstate – better random number generator; routines for changing generators

SYNOPSIS

long random()

srandom(seed)
int seed;

long *initstate(seed, state, n)
unsigned seed;
long *state;
int n;

long *setstate(state)
long *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*(3C) 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*(3C), 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 previous state array; its 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.

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(3C)

BUGS

About 2/3 the speed of *rand*(3C).

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), egrep(1), fgrep(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

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). The third parameter is a pointer to a routine which is called with a pointer to a directory entry and should return a non zero value if the directory entry should be included in the array. If this pointer is null, then all the directory entries will be included. The last argument is a pointer to a routine 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 will sort the array alphabetically.

Scandir returns the number of entries in the array and a pointer to the array through the parameter namelist.

SEE ALSO

directory(3), malloc(3), qsort(3)

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

setjmp, longjmp – non-local goto

SYNOPSIS

#include <setjmp.h>

val = setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;

val = _setjmp(env)
jmp_buf env;

_longjmp(env, val)
jmp_buf env;

DESCRIPTION

Setjmp and *longjmp* 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*. *Setjmp* also saves the register environment. *Setjmp* returns the value 0. If a *longjmp* call will be made, the routine which called *setjmp* should not return until after the *longjmp* has returned control (see below).

Longjmp restores the environment saved by the last call of *setjmp*, and 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*. The calling function must not itself have returned in the interim, otherwise *longjmp* will be returning control to a possibly non-existent environment. All memory-bound data have values as of the time *longjmp* was called. The machine registers are restored to the values they had at the time that *setjmp* was called. But, because the **register** storage class is only a hint to the C compiler, variables declared as **register** variables may not necessarily be assigned to machine registers, so their values are unpredictable after a *longjmp*. This is especially a problem for programmers trying to write machine-independent C routines.

The following code fragment indicates the flow of control of the *setjmp* and *longjmp* combination:

```
. . . function declaration
        jmp_buf         my_environment;

         . . . code . . .
        if (setjmp(my_environment)) {
                this is the code after the return from longjmp
                 . . . more code . . . .
                register variables have unpredictable values
                 . . . more code . . . .
        } else {
                this is the return from setjmp
                 . . . more code . . . .
                Do not modify register variables
                in this leg of the code
                 . . . more code . . . .
        }
```

Setjmp and *longjmp* save and restore the signal mask *sigsetmask*(2), while *_setjmp* and *_longjmp* manipulate only the C stack and registers.

SEE ALSO

sigsetmask(2), sigvec(2), signal(3)

BUGS

Setjmp does not save current notion of whether the process is executing on the signal stack. The result is that a longjmp to some place on the signal stack leaves the signal stack state incorrect.

NAME

setuid, seteuid, setruid, setgid, setegid, setrgid – set user and group ID

SYNOPSIS

setuid(uid)
seteuid(euid)
setruid(ruid)

setgid(gid)
setegid(egid)
setrgid(rgid)

DESCRIPTION

Setuid (*setgid*) sets both the real and effective user ID (group ID) of the current process to as specified.

Seteuid (*setegid*) sets the effective user ID (group ID) of the current process.

Setruid (*setruid*) sets the real user ID (group ID) of the current process.

These calls are only permitted to the super-user or if the argument is the real or effective ID.

SEE ALSO

setreuid(2), setregid(2), getuid(2), getgid(2)

DIAGNOSTICS

Zero is returned if the user (group) ID is set; –1 is returned otherwise, with the global variable *errno* set as for setreuid or setregid.

NAME

signal – simplified software signal facilities

SYNOPSIS

#include <signal.h>

(*signal(sig, func))()
void (*func)();

DESCRIPTION

Signal is a simplified interface to the more general *sigvec*(2) facility.

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 changed

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 occurences 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* 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. An *execve*(2) resets all caught signals to the default action; ignored signals remain ignored.

RETURN VALUE

The previous action is returned on a successful call. Otherwise, –1 is returned and *errno* is set to indicate the error.

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).

SEE ALSO

kill(1), ptrace(2), kill(2), sigvec(2), sigblock(2), sigsetmask(2), sigpause(2), sigstack(2), setjmp(3), tty(4)

NOTES (VAX-11)

The handler routine can be declared:

```
handler(sig, code, scp)
```

Here *sig* is the signal number, into which the hardware faults and traps are mapped as defined below. Code is a parameter which is either a constant as given below or, for compatibility mode faults, the code provided by the hardware. *Scp* is a pointer to the *struct sigcontext* used by the system to restore the process context from before the signal. Compatibility mode faults are distinguished from the other SIGILL traps by having PSL_CM set in the psl.

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
Decimal overflow trap	SIGFPE	FPE_DECOVF_TRAP
Subscript-range	SIGFPE	FPE_SUBRNG_TRAP
Floating overflow fault	SIGFPE	FPE_FLTOVF_FAULT
Floating divide by zero fault	SIGFPE	FPE_FLTDIV_FAULT
Floating underflow fault	SIGFPE	FPE_FLTUND_FAULT
Length access control	SIGSEGV	
Protection violation	SIGBUS	
Reserved instruction	SIGILL	ILL_RESAD_FAULT

Customer-reserved instr.	SIGEMT	
Reserved operand	SIGILL	ILL_PRIVIN_FAULT
Reserved addressing	SIGILL	ILL_RESOP_FAULT
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	
Compatibility-mode	SIGILL	hardware supplied code
Chme	SIGSEGV	
Chms	SIGSEGV	
Chmu	SIGSEGV	

NAME

sleep – suspend execution for interval

SYNOPSIS

sleep(seconds)
unsigned seconds;

DESCRIPTION

Sleep suspends the current process from execution for the number of seconds specified by the argument. The actual suspension time may be up to 1 second less than that requested, because scheduled wakeups occur at fixed 1-second intervals, and may be an arbitrary amount longer because of other activity in the system.

Sleep is implemented by setting an interval timer and pausing until it expires. The previous state of this timer is saved and restored. If the sleep time exceeds the time to the expiration of the previous value of the timer, the process sleeps only until the timer would have expired, and the signal which occurs with the expiration of the timer is sent one second later.

SEE ALSO

setitimer(2), sigpause(2)

BUGS

An interface with finer resolution is needed.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, index, rindex – string operations

SYNOPSIS

```
#include <strings.h>

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

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

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

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

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

char *strncpy(s1, s2, n)
char *s1, *s2;

strlen(s)
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.

Strcat appends a copy of string *s2* to the end of string *s1*. *Strncat* copies at most *n* characters. Both return a pointer to the null-terminated result.

Strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, according as *s1* is lexicographically greater than, equal to, or less than *s2*. *Strncmp* makes the same comparison but looks at at most *n* characters.

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 return *s1*.

Strlen returns the number of non-null characters in *s*.

Index (*rindex*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or zero if *c* does not occur in the string.

BUGS

Strcmp uses native character comparison, which is signed on the Sun.

On the Sun processor (and on some other machines), you can *NOT* use a zero pointer to indicate a null string. A zero pointer is an error and results in an abort of the program. If you wish to indicate a null string, you must have a pointer that points to an explicit null string. On PDP-11's and VAX'en, a source pointer of zero (0) can generally be used to indicate a null string. Programmers using NULL to represent an empty string should be aware of this portability issue.

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 high-ender machines (IBM 360's, MC68000's, etc) and low-ender machines (PDP-11's and VAX'es).**

***Nbytes* should be even.**

The *from* and *to* addresses should not overlap in portable programs.

NAME

syslog, openlog, closelog – control system log

SYNOPSIS

#include <syslog.h>

openlog(ident, logstat)
char *ident;

syslog(priority, message, parameters ...)
char *message;

closelog()

DESCRIPTION

Syslog arranges to write the *message* onto the system log maintained by *syslog*(8). The message is tagged with *priority*. The message looks like a *printf*(3S) 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 *syslog*(8) and output to the system console or files as appropriate.

If special processing is needed, *openlog* can be called to initialize the log file. Parameters are *ident* which is prepended to every message, and *logstat* which is a bit field indicating special status; current values are:

LOG_PID log the process id with each message: useful for identifying instantiations of daemons.

Openlog returns zero on success. If *syslog* cannot send datagrams to *syslog*(8), then it writes on */dev/console* instead. If */dev/console* cannot be written, standard error is used. In either case, it returns -1.

Closelog can be used to close the log file. It is automatically closed on a successful exec system call (see *execve*(2)).

EXAMPLES

```
syslog(LOG_SALERT, "who: internal error 23");

openlog("serverftp", LOG_PID);
syslog(LOG_INFO, "Connection from host %d", CallingHost);
```

SEE ALSO

syslog(8)

NAME

system – issue a shell command

SYNOPSIS

system(string)
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

ttyname, isatty, ttyslot – find name of a terminal

SYNOPSIS

char *ttyname(filedes)

isatty(filedes)

ttyslot()

DESCRIPTION

Ttyname returns a pointer to the null-terminated path name of the terminal device associated with file descriptor *filedes*.

Isatty returns 1 if *filedes* is associated with a terminal device, 0 otherwise.

Ttyslot returns the number of the entry in the *ttys*(5) file for the control terminal of the current process.

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'.

Ttyslot returns 0 if '/etc/ttys' is inaccessible or if it cannot determine the control terminal.

BUGS

The return value points to static data whose content is overwritten by each call.

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*(3S)) 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

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
getopt	getopt.3c	get option letter from argv
gtty	stty.3c	set and get terminal state
nice	nice.3c	set program priority
optarg	getopt.3c	get option letter from argv
optind	getopt.3c	get option letter from argv
pause	pause.3c	stop until signal
rand	rand.3c	random number generator
srand	rand.3c	random number generator
stty	stty.3c	set and get terminal state
time	time.3c	get date and time
times	times.3c	get process times
tmpnam	tmpnam.3c	create a name for a temporary file
ulimit	ulimit.3c	get and set user limits
utime	utime.3c	set file times
vlimit	vlimit.3c	control maximum system resource consumption
vtimes	vtimes.3c	get information about resource utilization

NAME

alarm – schedule signal after specified time

SYNOPSIS

alarm(seconds)
unsigned seconds;

DESCRIPTION

This interface is obsoleted by setitimer(2).

Alarm causes signal SIGALRM, see *sigvec*(2), 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.

The return value is the amount of time previously remaining in the alarm clock.

SEE ALSO

sigpause(2), sigvec(2), signal(3), sleep(3)

NAME

getopt, optarg, optind – get option letter from argv

SYNOPSIS

```
int getopt(argc, argv, optstring)
int argc;
char **argv;
char *optstring;

extern char *optarg;
extern int optind;
```

DESCRIPTION

This routine is included for compatibility with UNIX system-III. It is of marginal value, and should not be used in new programs.

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':
                    infile = optarg;
                    break;
                case 'o':
                    ofile = optarg;
                    bufsiza = 512;
                    break;
                case '?':
                    errflg++;
                }
        if (errflg) {
                fprintf(stderr, "usage: . . . ");
                exit(2);
        }
        for (; optind < argc; optind++) {
                if (access(argv[optind], 4)) {
        .
        .
        .
}
```

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 super-user. The priority is limited to the range –20 (most urgent) to 20 (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, *nice* should be called successively with arguments –40 (goes to priority –20 because of truncation), 20 (to get to 0), then 0 (to maintain compatibility with previous versions of this call).

SEE ALSO

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

NAME

pause – stop until signal

SYNOPSIS

pause()

DESCRIPTION

This function is obsoleted by sigpause(2).

Pause never returns normally. It is used to give up control while waiting for a signal from *kill*(2) or an interval timer, see *setitimer*(2). 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)

NAME

rand, srand – random number generator

SYNOPSIS

srand(seed)
int seed;

rand()

DESCRIPTION

The newer random(3) should be used in new applications; rand remains for compatibility.

Rand uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$.

The generator is reinitialized by calling *srand* with 1 as argument. It can be set to a random starting point by calling *srand* with whatever you like as argument.

SEE ALSO

random(3)

BUGS

The low bits of the numbers generated are not very random; use the middle bits. In particular the lowest bit is deterministically alternatingly 0 and 1.

NAME

stty, gtty – set and get terminal state

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

time, ftime – get date and time

SYNOPSIS

```
timeofday = time(0)

timeofday = time(tloc)
long *tloc;

#include <sys/types.h>
#include <sys/timeb.h>
ftime(tp)
struct timeb *tp;
```

DESCRIPTION

These interfaces are obsoleted by gettimeofday(2).

Time 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.

The *ftime* entry fills in a structure pointed to by its argument, as defined by *<sys/timeb.h>*:

```
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), settimeofday(2), ctime(3)

NAME

times – get process times

SYNOPSIS

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

times(buffer)
struct tms *buffer;
```

DESCRIPTION

This interface is obsoleted by getrusage(2).

Times returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/HZ seconds, where HZ is 60.

This is the structure returned by *times*:

```
struct tms {
        time_t  tms_utime;          /* user time */
        time_t  tms_stime;          /* system time */
        time_t  tms_cutime;         /* user time, children */
        time_t  tms_cstime;         /* system time, children */
};
```

The children times are the sum of the children's process times and their children's times.

SEE ALSO

time(1), getrusage(2), wait3(2), time(3C)

NAME

tmpnam – create a name for a temporary file

SYNOPSIS

#include <stdio.h>

char *tmpnam(s)
char *s;

DESCRIPTION

This routine is included for system-III compatibility.

Tmpnam generates a file name that can safely be used for a temprary file. If (int)*s* is zero, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If (int)*s* is nonzero, *s* is assumed to be the address of an array of at least **L_tmpnam** bytes; *tmpnam* places its result in that array and returns *s* as its value.

Tmpnam generates a different file name each time it is called.

Files created using *tmpnam* and either *fopen* or *creat* are only temporary in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *unlink*(2) to remove the file when its use is ended.

SEE ALSO

creat(2), unlink(2), mktemp(3), fopen(3S)

BUGS

If called more than 17,576 times in a single process, *tmpnam* will start recycling previously used names.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using *tmpnam* or *mktemp*, and the file names are chosen so as to render duplication by other means unlikely.

NAME

ulimit – get and set user limits

SYNOPSIS

long ulimit(cmd, newlimit)
int cmd;

DESCRIPTION

This function is included for system-III compatibility, and is obsoleted by *setrlimit*(2).

This function provides for control over process limits. The *cmd* values available are:

1 Get the process's file size limit. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

2 Set the process's file size limit to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. *Ulimit* will fail and the limit will be unchanged if a process with an effective user ID other than the super-user attempts to increase its file size limit.

3 Get the maximum possible break value. See *brk*(2).

RETURN VALUE

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

SEE ALSO

brk(2), setrlimit(2), write(2)

NAME

utime – set file times

SYNOPSIS

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

utime(file, timep)
char *file;
time_t timep[2];
```

DESCRIPTION

This interface is obsoleted by utimes(2).

The *utime* call uses the 'accessed' and 'updated' times in that order from the *timep* vector to set the corresponding recorded times for *file*.

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.

SEE ALSO

utimes(2), stat(2)

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*.

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 <*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 512 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

getrusage(2), wait3(2)

NAME

intro – introduction to mathematical library functions

DESCRIPTION

These functions constitute the math library, *libm*. They are automatically loaded as needed by the Fortran compiler *f77*(1). 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	trigonometric functions
asin	sin.3m	trigonometric functions
atan	sin.3m	trigonometric functions
atan2	sin.3m	trigonometric functions
cabs	hypot.3m	Euclidean distance
ceil	floor.3m	absolute value, floor, ceiling functions
cos	sin.3m	trigonometric functions
cosh	sinh.3m	hyperbolic functions
exp	exp.3m	exponential, logarithm, power, square root
fabs	floor.3m	absolute value, floor, ceiling functions
floor	floor.3m	absolute value, floor, ceiling functions
gamma	gamma.3m	log gamma function
hypot	hypot.3m	Euclidean distance
j0	j0.3m	bessel functions
j1	j0.3m	bessel functions
jn	j0.3m	bessel functions
log	exp.3m	exponential, logarithm, power, square root
log10	exp.3m	exponential, logarithm, power, square root
pow	exp.3m	exponential, logarithm, power, square root
sin	sin.3m	trigonometric functions
sinh	sinh.3m	hyperbolic functions
sqrt	exp.3m	exponential, logarithm, power, square root
tan	sin.3m	trigonometric functions
tanh	sinh.3m	hyperbolic functions
y0	j0.3m	bessel functions
y1	j0.3m	bessel functions
yn	j0.3m	bessel functions

NAME

exp, log, log10, pow, sqrt – exponential, logarithm, power, square root

SYNOPSIS

```
#include <math.h>

double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

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

double sqrt(x)
double x;
```

DESCRIPTION

Exp returns the exponential function of *x*.

Log returns the natural logarithm of *x*; *log10* returns the base 10 logarithm.

Pow returns x^y.

Sqrt returns the square root of *x*.

SEE ALSO

hypot(3M), sinh(3M), intro(2)

DIAGNOSTICS

Exp and *pow* return a huge value when the correct value would overflow; *errno* is set to ERANGE. *Pow* returns 0 and sets *errno* to EDOM when the second argument is negative and non-integral and when both arguments are 0.

Log returns 0 when *x* is zero or negative; *errno* is set to EDOM.

Sqrt returns 0 when *x* is negative; *errno* is set to EDOM.

NAME

fabs, floor, ceil – absolute value, floor, ceiling functions

SYNOPSIS

```
#include <math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double x;
```

DESCRIPTION

Fabs returns the absolute value $| x |$.

Floor returns the largest integer not greater than x.

Ceil returns the smallest integer not less than x.

SEE ALSO

abs(3)

BUGS

The *fabs* function is actually in the standard C library, and should be moved to the math library.

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);
#ifdef vax
        if (y > 88.0)
#endif
#ifdef sun
        if (y > 706.0)
#endif
                error();
        y = exp(y);
        if(signgam)
                y = -y;
```

DIAGNOSTICS

A huge value is returned for negative integer arguments.

BUGS

There should be a positive indication of error.

NAME

hypot, cabs – Euclidean distance

SYNOPSIS

```
#include <math.h>

double hypot(x, y)
double x, y;

double cabs(z)
struct { double x, y;} z;
```

DESCRIPTION

Hypot and *cabs* return

sqrt(x*x + y*y),

taking precautions against unwarranted overflows.

SEE ALSO

exp(3M) for *sqrt*

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 arguments cause *y0, y1,* and *yn* to return a huge negative value and set *errno* to EDOM.

NAME

sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions

SYNOPSIS

```
#include &lt;math.h&gt;

double sin(x)
double x;

double cos(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

double atan2(x, y)
double x, y;
```

DESCRIPTION

Sin, cos and *tan* return trigonometric functions of radian arguments. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

Asin returns the arc sin in the range $-\pi/2$ to $\pi/2$.

Acos returns the arc cosine in the range 0 to π.

Atan returns the arc tangent of x in the range $-\pi/2$ to $\pi/2$.

Atan2 returns the arc tangent of x/y in the range $-\pi$ to π.

DIAGNOSTICS

Arguments of magnitude greater than 1 cause *asin* and *acos* to return value 0; *errno* is set to EDOM. The value of *tan* at its singular points is a huge number, and *errno* is set to ERANGE.

BUGS

The value of *tan* for arguments greater than about 2**31 is garbage.

NAME

sinh, cosh, tanh – hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh(x)

double cosh(x)
double x;

double tanh(x)
double x;
```

DESCRIPTION

These functions compute the designated hyperbolic functions for real arguments.

DIAGNOSTICS

Sinh **and** ***cosh*** **return a huge value of appropriate sign when the correct value would overflow.**

NAME

intro – introduction to network library functions

DESCRIPTION

This section describes functions that are applicable to the DARPA Internet network, which are part of the standard C library.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
endhostent	gethostent.3n	get network host entry
endnetent	getnetent.3n	get network entry
endprotoent	getprotoent.3n	get protocol entry
endservent	getservent.3n	get service entry
gethostbyaddr	gethostent.3n	get network host entry
gethostbyname	gethostent.3n	get network host entry
gethostent	gethostent.3n	get network host entry
getnetbyaddr	getnetent.3n	get network entry
getnetbyname	getnetent.3n	get network entry
getnetent	getnetent.3n	get network entry
getprotobyname	getprotoent.3n	get protocol entry
getprotobynumber	getprotoent.3n	get protocol entry
getprotoent	getprotoent.3n	get protocol entry
getservbyname	getservent.3n	get service entry
getservbyport	getservent.3n	get service entry
getservent	getservent.3n	get service entry
htonl	byteorder.3n	convert values between host and network byte order
htons	byteorder.3n	convert values between host and network byte order
inet_addr	inet.3n	Internet address manipulation
inet_lnaof	inet.3n	Internet address manipulation
inet_makeaddr	inet.3n	Internet address manipulation
inet_netof	inet.3n	Internet address manipulation
inet_network	inet.3n	Internet address manipulation
inet_ntoa	inet.3n	Internet address manipulation
ntohl	byteorder.3n	convert values between host and network byte order
ntohs	byteorder.3n	convert values between host and network byte order
rcmd	rcmd.3n	routines for returning a stream to a remote command
rexec	rexec.3n	return stream to a remote command
rresvport	rcmd.3n	routines for returning a stream to a remote command
ruserok	rcmd.3n	routines for returning a stream to a remote command
sethostent	gethostent.3n	get network host entry
setnetent	getnetent.3n	get network entry
setprotoent	getprotoent.3n	get protocol entry
setservent	getservent.3n	get service entry

NAME

htonl, htons, ntohl, ntohs – convert values between host and network byte order

SYNOPSIS

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

netlong = htonl(hostlong);
u_long netlong, hostlong;

netshort = htons(hostshort);
u_short netshort, hostshort;

hostlong = ntohl(netlong);
u_long hostlong, netlong;

hostshort = ntohs(netshort);
u_short hostshort, netshort;
```

DESCRIPTION

These routines convert 16 and 32 bit quantities between network byte order and host byte order. On machines such as the Sun these routines are defined as null macros in the include file *<netinet/in.h>*.

These routines are most often used in conjunction with Internet addresses and ports as returned by *gethostent*(3N) and *getservent*(3N).

SEE ALSO

gethostent(3N), getservent(3N)

BUGS

The VAX handles bytes backwards from most everyone else in the world. This is not expected to be fixed in the near future.

NAME

gethostent, gethostbyaddr, gethostbyname, sethostent, endhostent – get network host entry

SYNOPSIS

```
#include <netdb.h>

struct hostent *gethostent()

struct hostent *gethostbyname(name)
char *name;

struct hostent *gethostbyaddr(addr, len, type)
char *addr; int len, type;

sethostent(stayopen)
int stayopen

endhostent()
```

DESCRIPTION

Gethostent, *gethostbyname*, and *gethostbyaddr* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network host data base, */etc/hosts*.

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* address type */
        int     h_length;       /* length of address */
        char    *h_addr;        /* address */
};
```

The members of this structure are:

h_name Official name of the host.

h_aliases A zero terminated array of alternate names for the host.

h_addrtype The type of address being returned; currently always AF_INET.

h_length The length, in bytes, of the address.

h_addr A pointer to the network address for the host. Host addresses are returned in network byte order.

Gethostent reads the next line of the file, opening the file if necessary.

Sethostent opens and rewinds the file. If the *stayopen* flag is non-zero, the host data base will not be closed after each call to *gethostent* (either directly, or indirectly through one of the other "gethost" calls).

Endhostent closes the file.

Gethostbyname and *gethostbyaddr* sequentially search from the beginning of the file until a matching host name or host address is found, or until EOF is encountered. Host addresses are supplied in network order.

FILES

/etc/hosts

SEE ALSO

hosts(5)

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. Only the Internet address format is currently understood.

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent – get network entry

SYNOPSIS

#include <netdb.h>

struct netent *getnetent()

struct netent *getnetbyname(name)
char *name;

struct netent *getnetbyaddr(net, type)
long net;

setnetent(stayopen)
int stayopen

endnetent()

DESCRIPTION

Getnetent, *getnetbyname*, and *getnetbyaddr* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network data base, */etc/networks*.

```
struct  netent {
        char    *n_name;        /* official name of net */
        char    **n_aliases;    /* alias list */
        int     n_addrtype;     /* net number type */
        long    n_net;          /* net number */
};
```

The members of this structure are:

n_name The official name of the network.

n_aliases A zero terminated list of alternate names for the network.

n_addrtype The type of the network number returned; currently only AF_INET.

n_net The network number. Network numbers are returned in machine byte order.

Getnetent reads the next line of the file, opening the file if necessary.

Setnetent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getnetent* (either directly, or indirectly through one of the other "getnet" calls).

Endnetent closes the file.

Getnetbyname and *getnetbyaddr* sequentially search from the beginning of the file until a matching net name or net address is found, or until EOF is encountered. Network numbers are supplied in host order.

FILES

/etc/networks

SEE ALSO

networks(5)

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.

Only Internet network numbers are currently understood.

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent – get protocol entry

SYNOPSIS

```
#include &lt;netdb.h&gt;

struct protoent *getprotoent()

struct protoent *getprotobyname(name)
char *name;

struct protoent *getprotobynumber(proto)
int proto;

setprotoent(stayopen)
int stayopen

endprotoent()
```

DESCRIPTION

Getprotoent, *getprotobyname*, and *getprotobynumber* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol data base, */etc/protocols*.

```
struct  protoent {
        char    *p_name;        /* official name of protocol */
        char    **p_aliases;    /* alias list */
        long    p_proto;        /* protocol number */
};
```

The members of this structure are:

p_name The official name of the protocol.

p_aliases A zero terminated list of alternate names for the protocol.

p_proto The protocol number.

Getprotoent reads the next line of the file, opening the file if necessary.

Setprotoent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getprotoent* (either directly, or indirectly through one of the other "getproto" calls).

Endprotoent closes the file.

Getprotobyname and *getprotobynumber* sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

FILES

/etc/protocols

SEE ALSO

protocols(5)

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. Only the Internet protocols are currently understood.

NAME

getservent, getservbyport, getservbyname, setservent, endservent – get service entry

SYNOPSIS

```
#include <netdb.h>

struct servent *getservent()

struct servent *getservbyname(name, proto)
char *name, *proto;

struct servent *getservbyport(port, proto)
int port; char *proto;

setservent(stayopen)
int stayopen

endservent()
```

DESCRIPTION

Getservent, *getservbyname*, and *getservbyport* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, */etc/services*.

```
struct  servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        long    s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

s_name The official name of the service.

s_aliases A zero terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order.

s_proto The name of the protocol to use when contacting the service.

Getservent reads the next line of the file, opening the file if necessary.

Setservent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getservent* (either directly, or indirectly through one of the other "getserv" calls).

Endservent closes the file.

Getservbyname and *getservbyport* sequentially search from the beginning of the file until a matching protocol name or port number is found, or until EOF is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

FILES

/etc/services

SEE ALSO

getprotoent(3N), services(5)

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. Expecting port numbers to fit in a 32 bit quantity is probably naive.

NAME

inet_addr, inet_network, inet_makeaddr, inet_lnaof, inet_netof, inet_ntoa – Internet address manipulation

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

struct in_addr
inet_addr(cp)
char *cp;

int
inet_network(cp)
char *cp;

struct in_addr
inet_makeaddr(net, lna)
int net, lna;

int
inet_lnaof(in)
struct in_addr in;

int
inet_netof(in)
struct in_addr in;

char *
inet_ntoa(in)
struct in_addr in;
```

DESCRIPTION

The routines *inet_addr* and *inet_network* each interpret character strings representing numbers expressed in the Internet standard "." notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine *inet_makeaddr* takes an Internet network number and a local network address and constructs an Internet address from it. The routines *inet_netof* and *inet_lnaof* break apart Internet host addresses, returning the network number and local network address part, respectively.

The routine *inet_ntoa* returns a pointer to a string in the base 256 notation "d.d.d.d" described below.

All Internet address are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the "." notation take one of the following forms:

a.b.c.d
a.b.c
a.b
a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note that when an Internet address is viewed as a 32-bit integer quantity on the VAX the bytes referred to above appear as "d.c.b.a". That is, VAX bytes are ordered from right to left.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "128.net.host".

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as "net.host".

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as "parts" in a "." notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e. a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

SEE ALSO

gethostent(3N), getnetent(3N), hosts(5), networks(5),

DIAGNOSTICS

The value –1 is returned by *inet_addr* and *inet_network* for malformed requests.

BUGS

The problem of host byte ordering versus network byte ordering is confusing. A simple way to specify Class C network addresses in a manner similar to that for Class B and Class A is needed.

The return value from *inet_ntoa* points to static information which is overwritten in each call.

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 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 addresses consist of a port 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 *gethostent*(3N) routine, two user names and a flag indicating if the local user's name is the super-user. It then checks the files */etc/hosts.equiv* and, possibly, *.rhosts* in the current working directory (normally the local user's home directory) to see if the request for service is allowed. A 1 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 0. If the *superuser* flag is 1, the checking of the "host.equiv" file is bypassed.

SEE ALSO

rlogin(1C), rsh(1C), rexec(3N), rexecd(8C), rlogind(8C), rshd(8C)

BUGS

There is no way to specify options to the *socket* call which *rcmd* makes.

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(3N), rexecd(8C)

BUGS

There is no way to specify options to the *socket* call which *rexec* makes.

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 in-line 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. A *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, clrerr.*

SEE ALSO

open(2), close(2), read(2), write(2), fread(3S), fseek(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* (see *fclose*(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 integer 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 integer from stream
getchar	getc.3s	get character or integer from stream
gets	gets.3s	get a string from a stream
getw	getc.3s	get character or integer from stream
pclose	popen.3s	initiate I/O to/from a process
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
stdio	intro.3s	standard buffered input/output package
ungetc	ungetc.3s	push character back into input stream

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

#include <stdio.h>

fclose(stream)
FILE *stream;

fflush(stream)
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. The stream remains open.

SEE ALSO

close(2), fopen(3S), 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>

feof(stream)
FILE *stream;

ferror(stream)
FILE *stream

clearerr(stream)
FILE *stream

fileno(stream)
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.

Clrerr 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

fopen, freopen, fdopen – open a stream

SYNOPSIS

```
#include <stdio.h>

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

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

FILE *fdopen(fildes, type)
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.

Type is a character string having one of the following values:

"r" open for reading

"w" create for writing

"a" append: open for writing at end of file, or create for writing

In addition, each *type* may be followed by a '+' to have the file opened for reading and writing. "r+" positions the stream at the beginning of the file, "w+" creates or truncates it, and "a+" positions it at the end. 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 *type* of the stream must agree with the mode of the open file.

SEE ALSO

open(2), fclose(3S)

DIAGNOSTICS

Fopen and *freopen* return the pointer NULL if *filename* cannot be accessed.

BUGS

Fdopen is not portable to systems other than UNIX.

The read/write *types* do not exist on all systems. Those systems without read/write modes will probably treat the *type* as if the '+' was not present. These are unreliable in any event.

NAME

fread, fwrite – buffered binary input/output

SYNOPSIS

#include <stdio.h>

fread(ptr, sizeof(*ptr), nitems, stream)
FILE ***stream;**

fwrite(ptr, sizeof(*ptr), nitems, stream)
FILE ***stream;**

DESCRIPTION

Fread reads, into a block beginning at *ptr*, *nitems* of data of the type of **ptr* 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 *nitems* of data of the type of **ptr* beginning at *ptr* to the named output *stream*. It returns the number of items actually written.

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.

NAME

fseek, ftell, rewind – reposition a stream

SYNOPSIS

```
#include <stdio.h>

fseek(stream, offset, ptrname)
FILE *stream;
long offset;

long ftell(stream)
FILE *stream;

rewind(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 0, 1, or 2.**

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 UNIX; on some other systems it is a magic cookie, and the only foolproof way to obtain an *offset* for *fseek*.**

***Rewind*(*stream*) is equivalent to *fseek*(*stream*, 0L, 0).**

SEE ALSO

lseek(2), fopen(3S)

DIAGNOSTICS

Fseek **returns –1 for improper seeks.**

NAME

getc, getchar, fgetc, getw – get character or integer from stream

SYNOPSIS

#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar()

int fgetc(stream)
FILE *stream;

int getw(stream)
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 C **int** (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.

Data files written and read with *putw* and *getw* are not portable; the size of an **int** and the order in which data bytes are stored within an **int** varies between machines.

NAME

gets, fgets – get a string from a stream

SYNOPSIS

#include <stdio.h>

char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
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

popen, pclose – initiate I/O to/from a process

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 to filter *stdin*, and a type "w" to filter *stdout*.

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*(3S).

Popen always calls *sh*, never calls *csh*.

NAME

printf, fprintf, sprintf – formatted output conversion

SYNOPSIS

```
#include <stdio.h>

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

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

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

#include <varargs.h>
_doprnt(format, args, stream)
char *format;
va_list *args;
FILE *stream;
```

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'. All of these routines work by calling the implementation-dependent routine *_doprnt,* using the variable-length argument facilities of *varargs*(3).

Each of these functions converts, formats, and prints its arguments after the first 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 *arg*

Each conversion specification is introduced by the character **%**. Following the **%**, there may be

- an optional minus sign '–' which specifies *left adjustment* of the converted value in the indicated field;
- an optional digit string specifying a *field width;* if the converted value has fewer characters than the field width it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width; if the field width begins with a zero, zero-padding will be done instead of blank-padding;
- an optional period '**.**' which serves to separate the field width from the next digit string;
- an optional digit string specifying a *precision* which specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string;
- an optional '#' character specifying that the value should be converted to an "alternate form". For **c**, **d**, **s**, and **u**, conversions, this option has no effect. 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 non-zero 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 as they would otherwise be.
- the character **l** specifying that a following **d**, **o**, **x**, or **u** corresponds to a long integer *arg*.
- a character which indicates the type of conversion to be applied.

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

The conversion characters and their meanings are

dox The integer *arg* is converted to decimal, octal, or hexadecimal notation respectively.

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

e The float or double *arg* is converted in the style '[−]d.ddde± dd' where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced.

g The float or double *arg* is printed in style **d**, in style **f**, or in style **e**, whichever gives full precision in minimum space.

The %**e**, %**f**, and %**g** formats print IEEE indeterminate values (infinity or not-a-number) as "Infinity" or "Nan" respectively.

c The character *arg* is printed.

s *Arg* is taken to be a string (character pointer) and characters from the string 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.

u The unsigned integer *arg* is converted to decimal and printed (the result will be in the range 0 through MAXUINT, where MAXUINT equals 4294967295 on a VAX-11 or Sun and 65535 on a PDP-11).

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

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by *printf* are printed by *putc*(3S).

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:%02d", weekday, month, day, hour, min);
```

To print π to 5 decimals:

```
printf("pi = %.5f", 4*atan(1.0));
```

SEE ALSO

putc(3S), scanf(3S), ecvt(3)

BUGS

Very wide fields (>128 characters) fail.

The values "Infinity" and "Nan" cannot be read by *scanf*(3S).

NAME

putc, putchar, fputc, putw – put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc(c, stream)
char c;
FILE *stream;

putchar(c)

fputc(c, stream)
FILE *stream;

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

DESCRIPTION

Putc appends the character *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 C **int** (word) *w* to the output *stream*. It returns the integer 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++)" doesn't work sensibly.

Errors can occur long after the call to *putc*.

Data files written and read with *putw* and *getw* are not portable; the size of an **int** and the order in which data bytes are stored within an **int** varies between machines.

NAME

puts, fputs – put a string on a stream

SYNOPSIS

```
#include <stdio.h>

puts(s)
char *s;

fputs(s, stream)
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

scanf, fscanf, sscanf – formatted input conversion

SYNOPSIS

#include <stdio.h>

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

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

sscanf(s, format [, pointer] . . .)
char *s, *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. Blanks, tabs or newlines, 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 numerical maximum field width, and a conversion character.

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; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal:

% a single '%' is expected in the input at this point; no assignment is done.

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

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

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

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. The input field is terminated by a space character or a newline.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next non-space character, try '%1s'. 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 a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits possibly containing a decimal point, followed by an optional exponent field consisting of an E or e followed by an optionally signed integer.

[indicates a string not to be delimited by space characters. 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 string. If the first character is not circumflex (^), the input field is all characters until the first character not in the set between the brackets; if the first character after the left bracket is ^, the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

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

The *scanf* functions return the number of successfully matched and assigned input items. This can be used to decide how many input items were found. The constant EOF is returned upon end of input; note that this is different from 0, which means that no conversion was done; if conversion was intended, it was frustrated by an inappropriate character in the input.

For example, 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
```

will assign to *i* the value 25, *x* the value 5.432, and *name* will contain '*thompson\0*'. Or,

```
int i; float x; char name[50];
scanf("%2d%f%*d%[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to *i*, 789.0 to *x*, skip '0123', and place the string '56\0' in *name*. The next call to *getchar* will return 'a'.

SEE ALSO

atof(3), getc(3S), printf(3S)

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.

Scanf cannot read the strings which *printf*(3S) generates for IEEE indeterminate floating point values.

Scanf provides no way to convert a number in any arbitrary base (decimal, hex or octal) based on the traditional *C* conventions (leading 0 or 0x).

NAME

setbuf, setbuffer, setlinebuf – assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

setbuf(stream, buf)
FILE *stream;
char *buf;

setbuffer(stream, buf, size)
FILE *stream;
char *buf;
int size;

setlinebuf(stream)
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 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. If the standard stream **stderr** refers to a terminal it is line buffered.

Setbuf is used after a stream has been opened but before it is read or written. The character array *buf* is used instead of an automatically allocated buffer. If *buf* is the constant pointer NULL, input/output will be completely unbuffered. A manifest constant BUFSIZ tells how big an array is needed:

```
char buf[BUFSIZ];
```

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* (only) from block buffered or unbuffered to line buffered. Unlike *setbuf* and *setbuffer* it can be used at any time that the file descriptor 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.

SEE ALSO

fopen(3S), getc(3S), putc(3S), malloc(3), fclose(3S), puts(3S), printf(3S), fread(3S)

NAME

ungetc – push character back into input stream

SYNOPSIS

#include <stdio.h>

ungetc(c, stream)
FILE *stream;

DESCRIPTION

Ungetc pushes the character *c* back on 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 provided something has been read from the stream and the stream is actually buffered. Attempts to push EOF are rejected.

An *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

intro – introduction to other libraries

DESCRIPTION

This section contains manual pages describing other libraries, which are available only from C. The list below includes libraries which provide device independent plotting functions, terminal independent screen management routines for two dimensional non-bitmap display terminals, and functions for managing data bases with inverted indexes. All functions are located in separate libraries indicated in each manual entry.

FILES

/usr/lib/libcurses.a	screen management routines (see *curses*(3x))
/usr/lib/libdbm.a	data base management routines (see *dbm*(3x))
/usr/lib/libmp.a	multiple precision math library (see *mp*(3x))
/usr/lib/libplot.a	plot routines (see *plot*(3x))
/usr/lib/lib300.a	"
/usr/lib/lib300s.a	"
/usr/lib/lib450.a	"
/usr/lib/lib4014.a	"
/usr/lib/libtermcap.a	terminal handling routines (see *termcap*(3x))
/usr/lib/libtermcap_p.a	
/usr/lib/libtermlib.a	
/usr/lib/libtermlib_p.a	

NAME
curses – screen functions with "optimal" cursor motion

SYNOPSIS
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
ioctl(2), getenv(3), tty(4), termcap(5)

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 char 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) co-ordinates
inch()	get char at current (y,x) co-ordinates
initscr()	initialize screens
insch(c)	insert a char
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 char 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 char from *win*
wdeleteln(win)	delete line from *win*
werase(win)	erase *win*
wgetch(win)	get a char through *win*
wgetstr(win,str)	get a string through *win*
winch(win)	get char at current (y,x) in *win*
winsch(win,c)	insert char into *win*
winsertln(win)	insert line into *win*
wmove(win,y,x)	set current (y,x) co-ordinates 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

dbminit, fetch, store, delete, firstkey, nextkey – data base subroutines

SYNOPSIS

```
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

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*.

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.

There are no interlocks and no provisionunreliable cache flushing; thus concurrent updating and reading is risky.

NAME

itom, madd, msub, mult, mdiv, min, mout, pow, gcd, rpow – multiple precision integer arithmetic

SYNOPSIS

```
#include <mp.h>

madd(a, b, c)
MINT *a, *b, *c;

msub(a, b, c)
MINT *a, *b, *c;

mult(a, b, c)
MINT *a, *b, *c;

mdiv(a, b, q, r)
MINT *a, *b, *q, *r;

min(a)
MINT *a;

mout(a)
MINT *a;

pow(a, b, c, d)
MINT *a, *b, *c, *d;

gcd(a, b, c)
MINT *a, *b, *c;

rpow(a, n, b)
MINT *a, *b;
short n;

msqrt(a, b, r)
MINT *a, *b, *r;

sdiv(a, n, q, r)
MINT *a, *q;
short n, *r;

MINT *itom(n)
short n;
```

DESCRIPTION

These routines perform arithmetic on integers of arbitrary length. The integers are stored using the defined type *MINT*. Pointers to a *MINT* should be initialized using the function *itom*, which sets the initial value to *n*. After that space is managed automatically by the routines.

Madd, *msub* and *mult* assign to their third arguments the sum, difference, and product, respectively, of their first two arguments. *Mdiv* assigns the quotient and remainder, respectively, to its third and fourth arguments. *Sdiv* is like *mdiv* except that the divisor is an ordinary integer. *Msqrt* produces the square root and remainder of its first argument. *Rpow* calculates *a* raised to the power *b*, while *pow* calculates this reduced modulo *m*. *Min* and *mout* do decimal input and

output.

Use the –lmp loader option to obtain access to these functions. –lmp.

DIAGNOSTICS

Illegal operations and running out of memory produce messages and core images.

FILES

/usr/lib/libmp.a

NAME

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*(1G) filters
–l300	GSI 300 terminal
–l300s	GSI 300S terminal
–l450	DASI 450 terminal
–l4014	Tektronix 4014 terminal

SEE ALSO

plot(5), plot(1G), graph(1G)

FILES

/usr/lib/libplot.a
/usr/lib/lib300.a
/usr/lib/lib300s.a
/usr/lib/lib450.a
/usr/lib/lib4014.a

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 encoded output speed of the terminal as described in *tty*(4). 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), tty(4), termcap(5)

NAME

intro – introduction to special files and hardware support

DESCRIPTION

This section describes device interfaces to the operating system for disks, tapes, serial communications, high-speed network communications, and other devices such as mice, frame buffers and windows.

The operating system can be built with or without many of the devices listed here; we show for most devices the syntax in a description to *config*(8) to cause the device to be included in a system. For mose devices we also give a DIAGNOSTICS section which lists the error messages which the device may produce to appear on the system console, and in the system error log file /usr/adm/messages.

Section 4 has been broken up according to machine independent device interfaces, "4" entries, Sun specific devices "4S", Vax specific devices "4V", manual pages for protocol families "4F", and manual pages for protocols and raw interfaces "4P".

Most devices on the Sun workstation exist on the Multibus, whose common properties are described in *mb*(4S).

Devices which are present in every kernel include a driver for the paging drum *drum*(4), drivers for accessing physical, virtual and i/o memory *mem*(4S) and the drivers for the data sink /dev/null, *null*(4).

Communications lines are most often used with the terminal driver described in *tty*(4). The terminal driver runs on communications lines provided either by a communications driver such as *oct*(4S) or *zs*(4S) or on a more virtual terminal, either provided by the Sun console monitor *cons*(4S) or a true pseudo-terminal *pty*(4) used in applications such as windowing or remote networking.

Magnetic tapes all provide the interface described in *mtio*(4). Tape devices for the Sun include *ar*(4S) and *tm*(4S).

Disk controllers provide standard block and raw interfaces, as well as a set of ioctl's defined in *dkio*(4S) supporting disk formatting and bad block handling. Drivers available for the Sun include *xy*(4S) and *ip*(4S).

The operating system supports one or more *protocol families* supporting local network communications. The only complete protocol family in this version of the system is the Internet protocol family *inet*(4F). Each protocol family provides basic services to each protocol implementation such as packet fragmentation and reassembly, routing, addressing and basic transport. A protocol family is normally composed of a number of protocols, one per *socket*(2) type. It is not required that a protocol family support all socket types.

The primary network support is for the Internet protocol family described in *inet*(4F). Major protocols in this family include the Internet Protocol *ip*(4P) describing the universal datagram format, the stream Transport Control Protocol *tcp*(4P), the User Datagram Protocol *udp*(4P), the Address Resolution Protocol *arp*(4P), and the Internet Control Message Protocol *icmp*(4P). The primary network interface is for the 10 Megabit Ethernet *ec*(4S); a software loopback interface *lo*(4) also exists. General properties of these (and all) network interfaces are described in *if*(4N).

The general support in the system for local network routing is described in *routing*(4N); these facilities apply to all protocol families.

Miscellaneous devices include color frame buffers *cg**(4S), monochrome frame buffers *bw**(4S), the console frame buffer, *fb*(4S), the console mouse *mouse*(4S) and the window devices *win*(4S).

NAME

ar – Archive 1/4 inch Streaming Tape Drive

SYNOPSIS

device ar0 at mb0 csr 0x200 priority 3

DESCRIPTION

The Archive tape controller is a Sun 'QIC-II' interface to an Archive streaming tape drive. It provides a standard tape interface to the device, see *mtio*(4), with some deficiencies listed under BUGS below.

The maximum blocksize for the raw device is limited only by available memory.

FILES

/dev/rar0
/dev/nrar0   non-rewinding

SEE ALSO

mtio(4), tm(4S)
Archive Intelligent Tape Drive Theory of Operation, Archive Corporation (Sun 8000-1058-01)
Archive Product Manual (Sidewinder 1/4" Streaming Cartridge Tape Drive) (Sun 800-0628-01)
Sun 1/4" Tape Interface – User Manual (Sun 800-0415-01)

DIAGNOSTICS

ar%d: would not initialize.

ar%d: already open. The tape can be open by only one process at a time.

ar%d: no such drive.

ar%d: no cartridge in drive.

ar%d: cartridge is write protected.

ar: interrupt from unitialized controller %x.

ar%d: many retries, consider retiring this tape.

ar%d: %b error at block # %d punted.

ar%d: %b error at block # %d.

ar: giving up on Rdy, try again.

BUGS

The tape cannot reverse direction so BSF, BSR and FSR are not available.

The system will hang if the tape is removed while running.

When using the raw device, the number of bytes in any given transfer must be a multiple of 512 bytes. If it is not, the device driver returns an error.

NAME

arp – Address Resolution Protocol

SYNOPSIS

pseudo-device ether

DESCRIPTION

ARP is a protocol used to dynamically map between DARPA Internet and 10Mb/s Ethernet addresses. It is used by all the 10Mb/s Ethernet interface drivers.

ARP caches Internet-Ethernet address mappings. When an interface requests a mapping for an address not in the cache, ARP queues the message which requires the mapping and broadcasts a message on the associated network requesting the address mapping. If a response is provided, the new mapping is cached and any pending messages are transmitted. ARP will queue at most one packet while waiting for a mapping request to be responded to; only the most recently "transmitted" packet is kept.

To enable communications with systems which do not use ARP, ioctls are provided to enter and delete entries in the Internet-to-Ethernet tables. Usage:

```
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <net/if.h>
struct arpreq arpreq;

ioctl(s, SIOCSARP, (caddr_t)&arpreq);
ioctl(s, SIOCGARP, (caddr_t)&arpreq);
ioctl(s, SIOCDARP, (caddr_t)&arpreq);
```

Each ioctl takes the same structure as an argument. SIOCSARP sets an ARP entry, SIOCGARP gets an ARP entry, and SIOCDARP deletes an ARP entry. These ioctls may be applied to any socket descriptor *s*, but only by the super-user. The *arpreq* structure contains:

```
/*
 * ARP ioctl request
 */
struct arpreq {
        struct sockaddr arp_pa;         /* protocol address */
        struct sockaddr arp_ha;         /* hardware address */
        int     arp_flags;              /* flags */
};
/*  arp_flags field values */
#define ATF_COM         2       /* completed entry (arp_ha valid) */
#define ATF_PERM        4       /* permanent entry */
#define ATF_PUBL        8       /* publish (respond for other host) */
```

The address family for the *arp_pa* sockaddr must be AF_INET; for the *arp_ha* sockaddr it must be AF_UNSPEC. The only flag bits which may be written are ATF_PERM and ATF_PUBL. ATF_PERM causes the entry to be permanent if the ioctl call succeeds. The peculiar nature of the ARP tables may cause the ioctl to fail if more than 4 (permanent) Internet host addresses hash to the same slot. ATF_PUBL specifies that the ARP code should respond to ARP requests for the indicated host coming from other machines. This allows a Sun to act as an "ARP server" which may be useful in convincing an ARP-only machine to talk to a non-ARP machine.

ARP watches passively for hosts impersonating the local host (i.e. a host which responds to an ARP mapping request for the local host's address).

DIAGNOSTICS

duplicate IP address!! sent from ethernet address: %x:%x:%x:%x:%x:%x. ARP has discovered another host on the local network which responds to mapping requests for its own

Internet address.

SEE ALSO

ec(4S), ie(4S), inet(4F), arp(8C), ifconfig(8C)
An Ethernet Address Resolution Protocol, RFC826, Dave Plummer, MIT (Sun 800-1059-01)

BUGS

ARP packets on the Ethernet use only 42 bytes of data, however, the smallest legal Ethernet packet is 60 bytes (not including CRC). Some systems may not enforce the minimum packet size, others will.

NAME

bk – line discipline for machine-machine communication

SYNOPSIS

pseudo-device bk

DESCRIPTION

This line discipline provides a replacement for the tty driver *tty*(4) when high speed output to and especially input from another machine is to be transmitted over an asynchronous communications line. The discipline was designed for use by a (now obsolete) store-and-forward local network running over serial lines. It may be suitable for uploading of data from microprocessors into the system. If you are going to send data over asynchronous communications lines at high speed into the system, you must use this discipline, as the system otherwise may detect high input data rates on terminal lines and disable the lines; in any case the processing of such data when normal terminal mechanisms are involved saturates the system.

The line discipline is enabled by a sequence:

```
#include <sgtty.h>
int ldisc = NETLDISC, fildes; ...
ioctl(fildes, TIOCSETD, &ldisc);
```

A typical application program then reads a sequence of lines from the terminal port, checking header and sequencing information on each line and acknowledging receipt of each line to the sender, who then transmits another line of data. Typically several hundred bytes of data and a smaller amount of control information will be received on each handshake.

The old standard teletype discipline can be restored by doing:

```
ldisc = OTTYDISC;
ioctl(fildes, TIOCSETD, &ldisc);
```

While in networked mode, normal teletype output functions take place. Thus, if an 8 bit output data path is desired, it is necessary to prepare the output line by putting it into RAW mode using *ioctl*(2). This must be done **before** changing the discipline with TIOCSETD, as most *ioctl*(2) calls are disabled while in network line-discipline mode.

When in network mode, input processing is very limited to reduce overhead. Currently the input path is only 7 bits wide, with newline the only character terminating an input record. Each input record must be read and acknowledged before the next input is read as the system refuses to accept any new data when there is a record in the buffer. The buffer is limited in length, but the system guarantees to always be willing to accept input resulting in 512 data characters and then the terminating newline.

User level programs should provide sequencing and checksums on the information to guarantee accurate data transfer.

SEE ALSO

tty(4)

DIAGNOSTICS

None.

NAME

bwone – Sun one black and white frame buffer

SYNOPSIS

device bwone0 at mb0 csr 0xc0000 priority 3

DESCRIPTION

The *bwone* interface provides access to Sun-1 black-and-white graphics controller boards. It supports the FBIOGTYPE ioctl which a program can use to inquire as to the characteristics of the display device; see *fbio*(4S)

It supports the FBIOGPIXRECT ioctl which allows SunWindows to be run on it; see *fbio*(4S)

Reading or writing to the frame buffer is not allowed – you must use the *mmap*(2) system call to map the board into your address space.

FILES

/dev/bwone[0-9]

SEE ALSO

mmap(2), fb(4S), fbio(4S)
Sun 1024 Video Board – User Manual (Sun 800-0420)

DIAGNOSTICS

None.

BUGS

Use of vertical-retrace interrupts is not supported.

NAME

bwtwo – Sun two black and white frame buffer

SYNOPSIS

device bwtwo0 at mb0 csr 0x700000 priority 3

DESCRIPTION

The *bwtwo* interface provides access to Sun-2 Monochrome Video Controller boards. It supports the FBIOGTYPE ioctl which a program can use to inquire as to the characteristics of the display device; see *fbio*(4S)

It supports the FBIOGPIXRECT ioctl which allows SunWindows to be run on it; see *fbio*(4S)

Reading or writing to the frame buffer is not allowed – you must use the *mmap*(2) system call to map the board into your address space.

FILES

/dev/bwtwo[0-9]

SEE ALSO

mmap(2), fb(4S), fbio(4S)

DIAGNOSTICS

None.

BUGS

Use of vertical-retrace interrupts is not supported.

NAME

cgone – Sun-1 color graphics interface

SYNOPSIS

device cgone0 at mb0 csr 0xe8000 priority 3

DESCRIPTION

The *cgone* interface provides access to the Sun-1 color graphics controller board, which is normally supplied with a 13" or 19" RS170 color monitor. It provides the standard frame buffer interface as defined in *fbio*(4S).

It supports the FBIOGPIXRECT ioctl which allows SunWindows to be run on it; see *fbio*(4S)

The hardware consumes 16 kilobytes of Multibus memory space. The board starts at standard addresses 0xE8000 or 0xEC000. The board must be configured for interrupt level 3.

FILES

/dev/cgone[1-9]

SEE ALSO

mmap(2), fbio(4S)
Sun Color Video Board User's Manual (Sun 8000-0398, Rev B)
Barco GD33 Color Display 120VAC Operation Instructions (13") (Sun 800-1002-01)
Barco Color Display CD 252 120/220VAC Operation Guide (19") (Sun 800-1003-01)

DIAGNOSTICS

None.

BUGS

Use of color board vertical-retrace interrupts is not supported.

NAME

cons – driver for Sun console

SYNOPSIS

None; included in standard system.

DESCRIPTION

Cons **is an indirect driver for the Sun workstation console, which implements a standard UNIX terminal.** ***Cons*** **is implemented by calling the PROM resident monitor to perform I/O to and from the current system console, which is either a Sun frame buffer or an RS232 port.**

When the Sun window system ***win*****(4S) is active, console input is directed through the window system rather than being read from /dev/console.**

An ioctl TIOCCONS may be applied to serial devices other than the console to cause output which would normally appear on the console to instead be routed to the other devices. This is used by the window system which does a TIOCCONS on a pseudo-terminal to cause console output to be routed there rather than to the screen through the PROM monitor, since routing output through the PROM destroys the integrity of the screen.

FILES

/dev/console
/dev/ttya alternate console (serial port)

SEE ALSO

oct(4S), tty(4), zs(4S)

BUGS

TIOCCONS should be restricted to the owner of /dev/console.

NAME

dkio – generic disk control operations

DESCRIPTION

All Sun disk drivers support a set of ioctl's for disk formattting and labelling operations. Basic to these ioctl's are the definitions in <sun/dkio.h>:

```
/*
 * Structures and definitions for disk io control commands
 */

/* Disk identification */
struct dk_info {
        int     dki_ctlr;               /* controller address */
        short   dki_unit;               /* unit (slave) address */
        short   dki_ctype;              /* controller type */
        short   dki_flags;              /* flags */
};
/* controller types */
#define DKC_UNKNOWN     0
#define DKC_SMD2180     1
#define DKC_XY440       4
#define DKC_DSD5215     5
#define DKC_XY450       6
#define DKC_SCSI        7

/* flags */
#define DKI_BAD144    01        /* use DEC std 144 bad sector fwding */
#define DKI_MAPTRK 02           /* controller does track mapping */
#define DKI_FMTTRK 04           /* formats only full track at a time */
#define DKI_FMTVOL 0x08         /* formats only full volume at a time */

/* Definition of a disk's geometry */
struct dk_geom {
        unsigned short  dkg_ncyl;       /* # of data cylinders */
        unsigned short  dkg_acyl;       /* # of alternate cylinders */
        unsigned short  dkg_bcyl;       /* cyl offset (for fixed head area) */
        unsigned short  dkg_nhead;      /* # of heads */
        unsigned short  dkg_bhead;      /* head offset (for Larks, etc.) */
        unsigned short  dkg_nsect;      /* # of sectors per track */
        unsigned short  dkg_intrlv;     /* interleave factor */
        unsigned short  dkg_gap1;       /* gap 1 size */
        unsigned short  dkg_gap2;       /* gap 2 size */
        unsigned short  dkg_extra[10];  /* for compatible expansion */
};

/* Disk format request */
struct dk_fmt {
        daddr_t dkf_blkno;              /* starting block */
        daddr_t dkf_nblk;               /* # of blocks */
        u_char  dkf_fill;               /* fill data */
};

/* Disk re-map request */
```

```
struct dk_mapr {
        daddr_t dkm_fblk;            /* from block */
        daddr_t dkm_tblk;            /* to block */
        daddr_t dkm_nblk;            /* # blocks */
        u_char  dkm_fill;            /* fill data */
};

/* disk io control commands */
#define DKIOCHDR      _IO(d, 1)                    /* next I/O will read/write header */
#define DKIOCGGEOM    _IOR(d, 2, struct dk_geom)   /* Get geometry */
#define DKIOCSGEOM    _IOW(d, 3, struct dk_geom)   /* Set geometry */
#define DKIOCGPART    _IOR(d, 4, struct dk_map)    /* Get partition info */
#define DKIOCSPART    _IOW(d, 5, struct dk_map)    /* Set partition info */
#define DKIOCFMT      _IOW(d, 6, struct dk_fmt)    /* Format */
#define DKIOCMAP      _IOW(d, 7, struct dk_mapr)   /* Map */
#define DKIOCINFO     _IOR(d, 8, struct dk_info)   /* Get info */
```

The DKIOCGINFO ioctl returns a dk_info structure which tells the kind of the controller and attributes about how bad-block processing is done on the controller. Bad sectors can then be processed using either the DKIOCMAP request, which causes a sector to be re-mapped on the disk, or the DKIOCFMT request which causes a sector to be re-formatted. To read or write the header on a disk sector the DKIOCHDR ioctl can be used, it causes the next read or write request to also read or write the (drive-type-specific) header data.

The DKIOCGPART and DKIOCSPART get and set the controller's current notion of the partition table for the disk (without changing the partition table on the disk itself), while the DKIOCGGEOM and DKIOCSGEOM ioctl's do similar things for the per-drive geometry information. These can be used to format a drive, where the label does not exist before the drive is formatted.

SEE ALSO

ip(4S), xy(4S)

BUGS

The DKIOCMAP and DKIOCFMT request are incompletely implemented.

NAME
drum – paging device

SYNOPSIS
None; included with standard system.

DESCRIPTION
This file refers to the paging device in use by the system. This may actually be a subdevice of one of the disk drivers, but in a system with paging interleaved across multiple disk drives it provides an indirect driver for the multiple drives.

FILES
/dev/drum

BUGS
Reads from the drum are not allowed across the interleaving boundaries. Since these only occur every .5Mbytes or so, and since the system never allocates blocks across the boundary, this is usually not a problem.

NAME

ec – 3Com 10 Mb/s Ethernet interface

SYNOPSIS

device ec0 at mb0 csr 0xe0000 priority 3

DESCRIPTION

The *ec* interface provides access to a 10 Mb/s Ethernet network through a 3COM controller. For a general description of network interfaces see *if*(4N).

The hardware consumes 8 kilobytes of Multibus memory space. This memory is used for internal buffering by the board. The board starts at standard addresses 0xE0000 or 0xE2000. The board must be configured for interrupt level 3.

The interface software implements an exponential backoff algorithm when notified of a collision on the cable.

The interface handles the Internet protocol family, with the interface address maintained in Internet format. The Address Resolution Protocol *arp*(4P) is used to map 32-bit Internet addresses used in *inet*(4F) to the 48-bit addresses used on the Ethernet.

DIAGNOSTICS

ec%d: Ethernet jammed. After 16 failed transmissions and backoffs using the exponential backoff algorithm, the packet was dropped.

ec%d: can't handle af%d. The interface was handed a message with addresses formatted in an unsuitable address family; the packet was dropped.

SEE ALSO

arp(4N), if(4N), inet(4F)
3COM 3C400 Multibus Ethernet Controller Reference Manual (Sun 800-0398)

BUGS

The interface hardware is not capable of talking to itself, making diagnosis more difficult.

NAME

en – Sun 3 Mb/s experimental Ethernet interface

SYNOPSIS

device en0 at mb0 csr 0x100 priority 3

DESCRIPTION

The *en* interface provides access to a 3 Mb/s Ethernet network. The host's address is discovered at boot time by probing the Ethernet address register. For a general description of network interfaces, see *if*(4N).

The board consumes 256 bytes of Multibus I/O space starting at standard address 0x100.

The interface handles both Internet and PUP protocol families, with the interface address maintained in Internet format. PUP addresses are converted to Internet addresses by subsituting PUP network and host values for Internet network and imp values, and setting the Internet host number to zero.

DIAGNOSTICS

en%d: output error. The hardware indicated an error on the previous transmission.

en%d: send error. After 16 retransmissions the packet was dropped.

en%d: input error. The hardware indicated an error in reading a packet off the cable.

en%d: can't handle af%d. The interface was handed a message with addresses formatted in an unsuitable address family; the packet was dropped.

SEE ALSO

if (4N), inet(4F)
Sun 3Mbit Ethernet Board, User's Manual (Sun 800-0392)

BUGS

This hardware and driver are not supported.

NAME

fb – driver for Sun console frame buffer

SYNOPSIS

None; included in standard system.

DESCRIPTION

The *fb* driver provides indirect access to a Sun graphics controller board. It is an indirect driver for the Sun workstation console's frame buffer. At boot time, the workstation's frame buffer device is determined from information from the Monitor Proms and set to be the one that *fb* will indirect to. The device driver for the console's frame buffer must be configured into the kernel so that this indirect driver can access it.

The idea behind this driver is that user programs can open a known device, query its characteristics and access it in a device dependent way, depending on the type. *Fb* redirects *open*(2), *close*(2), *ioctl*(2), and *mmap*(2) calls to the real frame buffer. All of the Sun frame buffers support the same general interface; see *fbio*(4S)

FILES

/dev/fb

SEE ALSO

fbio(4S), bwone(4S), bwtwo(4S)

NAME

fbio – general properties of frame buffers

DESCRIPTION

All of the Sun frame buffers support the same general interface. Each responds to a FBIOGTYPE *ioctl*(2) which returns information in a structure defined in <sun/fbio.h>:

```
struct  fbtype {
        int     fb_type;        /* as defined below */
        int     fb_height;      /* in pixels */
        int     fb_width;       /* in pixels */
        int     fb_depth;       /* bits per pixel */
        int     fb_cmsize;      /* size of color map (entries) */
        int     fb_size;        /* total size in bytes */
};

#define FBTYPE_SUN1BW           0
#define FBTYPE_SUN1COLOR        1
#define FBTYPE_SUN2BW           2
```

Each device has a FBTYPE which is used by higher-level software to determine how to perform raster-op and other functions. Each device is used by opening it, doing a FBIOGTYPE *ioctl* to see which frame buffer type is present, and thereby selecting the appropriate device management routines.

Full fledged frame buffers, i.e., those that expect to run SunWindows, implement an FBIOGPIX-RECT *ioctl*(2), which returns a pixrect. This call is made only from inside the kernel. The returned pixrect is used by win(4S) for cursor tracking and colormap loading.

SEE ALSO

mmap(2), fb(4S), bwone(4S), bwtwo(4S), cgone(4S), win(4S)

NAME

icmp – Internet Control Message Protocol

SYNOPSIS

None; included automatically with *inet*(4F).

DESCRIPTION

The Internet Control Message Protocol ICMP is used by gateways and destination hosts which process datagrams to communicate errors in datagram processing to source hosts. (The datagram level of Internet is discussed in *ip*(4P).) ICMP uses the basic support of IP as if it were a higher level protocol, however ICMP is actually an integral part of IP.

ICMP messages are sent in several situations: for example when a datagram, cannot reach its destination, when the gateway does not have the buffering capacity to forward a datagram, and when the gateway can direct the host to send traffic on a shorter route.

The Internet protocol is not designed to be absolutely reliable. The purpose of these control messages is to provide feedback about problems in the communication environment, not to make IP reliable. There are still no guarantees that a datagram will be delivered or a control message will be returned. Some datagrams may still be undelivered without any report of their loss. The higher level protocols which use IP must implement their own reliability procedures if reliable communication is required.

The ICMP messages typically report errors in the processing of datagrams. To avoid the infinite regress of messages about messages etc., no ICMP messages are sent about ICMP messages. Also ICMP messages are only sent about errors in handling fragment 0 of fragmented datagrams.

There are 11 types of ICMP packets which can be received by the system. They are defined in this excerpt from <netinet/ip_icmp.h>, which also defines the values of some additional codes further specifying the cause of certain errors.

```
/*
 * Definition of type and code field values
 */
#define ICMP_ECHOREPLY              0         /* echo reply */
#define ICMP_UNREACH                3         /* dest unreachable, codes: */
#define         ICMP_UNREACH_NET              0       /* bad net */
#define         ICMP_UNREACH_HOST             1       /* bad host */
#define         ICMP_UNREACH_PROTOCOL         2       /* bad protocol */
#define         ICMP_UNREACH_PORT             3       /* bad port */
#define         ICMP_UNREACH_NEEDFRAG         4       /* IP_DF caused drop */
#define         ICMP_UNREACH_SRCFAIL          5       /* src route failed */
#define ICMP_SOURCEQUENCH           4         /* packet lost, slow down */
#define ICMP_REDIRECT               5         /* shorter route, codes: */
#define         ICMP_REDIRECT_NET       0             /* for network */
#define         ICMP_REDIRECT_HOST      1             /* for host */
#define         ICMP_REDIRECT_TOSNET    2             /* for tos and net */
#define         ICMP_REDIRECT_TOSHOST   3             /* for tos and host */
#define ICMP_ECHO                   8         /* echo service */
#define ICMP_TIMXCEED               11        /* time exceeded, code: */
#define         ICMP_TIMXCEED_INTRANS   0             /* ttl==0 in transit */
#define         ICMP_TIMXCEED_REASS     1             /* ttl==0 in reass */
#define ICMP_PARAMPROB              12        /* ip header bad */
#define ICMP_TSTAMP                 13        /* timestamp request */
#define ICMP_TSTAMPREPLY            14        /* timestamp reply */
#define ICMP_IREQ                   15        /* information request */
#define ICMP_IREQREPLY              16        /* information reply */
```

Arriving ECHO and TSTAMP packets cause the system to generate ECHOREPLY and TSTAMPREPLY packets. IREQ packets are not yet processed by the system, and are discarded. UNREACH, SOURCEQUENCH, TIMXCEED and PARAMPROB packets are processed internally by the protocols implemented in the system, or reflected to the user if a raw socket is being used; see *ip*(4P). REDIRECT, ECHOREPLY, TSTAMPREPLY and IREQREPLY are also reflected to users of raw sockets. In addition, REDIRECT messages cause the kernel routing tables to be updated; see *routing*(4N).

SEE ALSO

inet(4F), ip(4P)
Internet Control Message Protocol, RFC792, J. Postel, USC-ISI (Sun 800-1064-01)

BUGS

IREQ messages are not processed properly: the address fields are not set.

Messages which are source routed are not sent back using inverted source routes, but rather go back through the normal routing mechanisms.

NAME

if – general properties of network interfaces

DESCRIPTION

Each network interface in a system corresponds to a path through which messages may be sent and received. A network interface usually has a hardware device associated with it, though certain interfaces such as the loopback interface, *lo*(4), do not.

At boot time each interface which has underlying hardware support makes itself known to the system during the autoconfiguration process. Once the interface has acquired its address it is expected to install a routing table entry so that messages may be routed through it. Most interfaces require some part of their address specified with an SIOCSIFADDR ioctl before they will allow traffic to flow through them. On interfaces where the network-link layer address mapping is static, only the network number is taken from the ioctl; the remainder is found in a hardware specific manner. On interfaces which provide dynamic network-link layer address mapping facilities (e.g. 10Mb/s Ethernets using *arp*(4P),), the entire address specified in the ioctl is used.

The following *ioctl* calls may be used to manipulate network interfaces. Unless specified otherwise, the request takes an *ifreq* structure as its parameter. This structure has the form

```
struct  ifreq {
        char    ifr_name[16];           /* name of interface (e.g. "ec0") */
        union {
                struct  sockaddr ifru_addr;
                struct  sockaddr ifru_dstaddr;
                short   ifru_flags;
        } ifr_ifru;
#define ifr_addr        ifr_ifru.ifru_addr      /* address */
#define ifr_dstaddr     ifr_ifru.ifru_dstaddr   /* other end of p-to-p link */
#define ifr_flags       ifr_ifru.ifru_flags     /* flags */
};
```

SIOCSIFADDR
: Set interface address. Following the address assignment, the "initialization" routine for the interface is called.

SIOCGIFADDR
: Get interface address.

SIOCSIFDSTADDR
: Set point to point address for interface.

SIOCGIFDSTADDR
: Get point to point address for interface.

SIOCSIFFLAGS
: Set interface flags field. If the interface is marked down, any processes currently routing packets through the interface are notified.

SIOCGIFFLAGS
: Get interface flags.

SIOCGIFCONF
: Get interface configuration list. This request takes an *ifconf* structure (see below) as a value-result parameter. The *ifc_len* field should be initially set to the size of the buffer pointed to by *ifc_buf*. On return it will contain the length, in bytes, of the configuration list.

```
/*
 * Structure used in SIOCGIFCONF request.
 * Used to retrieve interface configuration
```

```
 * for machine (useful for programs which
 * must know all networks accessible).
 */
struct  ifconf {
        int     ifc_len;         /* size of associated buffer */
        union {
                caddr_t ifcu_buf;
                struct  ifreq *ifcu_req;
        } ifc_ifcu;
#define ifc_buf ifc_ifcu.ifcu_buf /* buffer address */
#define ifc_req ifc_ifcu.ifcu_req /* array of structures returned */
};
```

SEE ALSO

arp(4P), ec(4S), en(4S), lo(4)

NAME

inet – Internet protocol family

SYNOPSIS

options INET
pseudo-device inet

DESCRIPTION

The Internet protocol family is a collection of protocols layered atop the *Internet Protocol* (IP) transport layer, and utilizing the Internet address format. The Internet family provides protocol support for the SOCK_STREAM, SOCK_DGRAM, and SOCK_RAW socket types; the SOCK_RAW interface provides access to the IP protocol.

ADDRESSING

Internet addresses are four byte quantities, stored in network standard format (on the VAX these are word and byte reversed). The include file <*netinet/in.h*> defines this address as a discriminated union.

Sockets in the Internet protocol family utilize the following addressing structure,

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

(Library routines to return and manipulate structures of this form are in section 3N of the manual; see *intro*(3N) and the other section 3 entries mentioned under SEE ALSO below.) Each socket has a local address specifiable in this form, which can be established with *bind*(2); the *getsockname*(2) call returns this address. Each socket also may be bound to a peer socket with an address specified in this form; this peer address can be specified in a *connect*(2) call, or transiently with a single message in a *sendto* or *sendmsg* call; see *send*(2). The peer address of a socket is returned by the *getpeername*(2) call.

The sin_addr field of the socket address specifies the Internet address of the machine on which the socket is located. A special value may be specified or returned for this field, sin_addr.s_addr==INADDR_ANY. This address is a "wildcard" and matches any of the legal internet addresses on the local machine. This address is useful when a process neither knows (nor cares) what the local Internet address is, but even more useful for server processes with which to service all requests to the current machine. Since a machine can have several addresses (one per hardware network interface), specifying a single address would restrict access to the service to those clients which specified the address of that interface. By specifying INADDR_ANY, the server can arrange to service clients from all interfaces.

When a socket address is bound, the networking system checks that there is an interface with the address specified available on the current machine (unless, of course, a wildcard address is specified), and returns an error EADDRNOTAVAIL if no such interface is found.

The local port address specified in a *bind*(2) call is restricted to be greater than IPPORT_RESERVED (=1024, in <netinet/in.h>) unless the creating process is running as the super-user, providing a space of protected port numbers. The local port address is also required to not be in use in order for it to be assigned. This is checked by looking for another socket of the same type which has the same local address and local port number. If such a socket already exists, you will not be able to create another socket at the same address, and will instead get the error EADDRINUSE. If the local port address is specified as 0, then the system picks a unique port address not less than IPPORT_RESERVED and assigns it to the port. A unique local port address is also picked for a socket which is not bound but which is used with *connect*(2) or *sendto*(2); this allows *tcp*(4p) connections to be made by simply doing *socket*(2) and then *connect*(2) in the case where the local port address is not significant; it is defaulted by the system. Similarly if you are sending datagrams with *udp*(4P) and do not care which port they come from, you can just do *socket*(2) and *sendto*(2) and let the system pick a port number.

Let us say that two sockets are incompatible if they have the same port number, are not conected to other sockets, and do not have different local host addresses. (It is possible to have two sockets with the same port number and different local host addresses because a machine may have several local addresses from its different network interfaces.) The Internet system does not allow such incompatible sockets to exist on a single machine. Consider a socket which has a specific local host and local port number on the current machine. If another process tries to create a socket with a wildcard local host address and the same port number then that request will be denied. For connection based sockets this prevents these two sockets from attempting to connect to the same foreign host/socket, and thereby causing great havoc. For connectionless sockets this prevents the dilemma which would result from trying to determine who to deliver an incoming datagram to (since more than one socket could match an address given on a datagram). The same restriction applies if the wildcard socket exists first. (If both sockets are wildcard, then the normal restrictions on duplicate addresses apply.)

A socket option SO_REUSEADDR exists to allow incompatible sockets to be created. This option is needed to implement the File Transfer Protocol (FTP) which requires that a connection be made from an existing port number (the port number of its primary connection) to a different port number on the same remote host. The danger here is that the user would attempt to connect this second port to the same remote host/port that the primary connection was using. In using SO_REUSEADDR the user is pledging not to do this, since this will cause the first connection to abort.

When a *connect*(2) is done, the Internet system first checks that the socket is not already connected. If does not allow connections to port number 0 on another host, nor does it allow connections to a wildcard host (sin_addr.s_addr==INADDR_ANY); attempts to do this yield EADDRINUSE. If the socket from which the connection is being made currently has a wildcard local address (either because it was bound to a specific port with a wildcard address, or was never subjected to *bind*(2)), then the system picks a local Internet address for the socket from the set of addresses of interfaces on the local machine. If there is an interface on the local machine on the same network as the machine being connected to, then that address is used. Otherwise, the "first" local network interface is used (this is the one that prints out first in "netstat –i"; see *netstat*(8)). Although it is not supposed to matter which interface address is used, in practice it would probably be better to select the address of the interface through which the packets are to be routed. This is not currently done (as it would involve a fair amount of additional overhead for datagram transmission).

PROTOCOLS

The Internet protocol family supported by the operating system is comprised of the Internet Datagram Protocol (IP) *ip*(4P), Address Resolution Protocol (ARP) *arp*(4P), Internet Control Message Protocol (ICMP) *icmp*(4P), Transmission Control Protocol (TCP) *tcp*(4P), and User Datagram Protocol (UDP) *udp*(4P).

TCP is used to support the SOCK_STREAM abstraction while UDP is used to support the SOCK_DGRAM abstraction. A raw interface to IP is available by creating an Internet socket of type SOCK_RAW; see *ip*(4P). The ICMP message protocol is not directly accessible, and is used by the system to handle and report errors in protocol processing. The ARP protocol is used to translate 32-bit Internet host numbers into the 48 bit addresses needed for an Ethernet.

SEE ALSO

intro(3N), byteorder(3N), gethostent(3N), getnetent(3N), getprotoent(3N), getservent(3N), inet(3N), network(3N), arp(4P), tcp(4P), udp(4P), ip(4P)
Internet Protocol Transition Workbook, Network Information Center, SRI (Sun 800-1056-01)

Internet Protocol Implementation Guide, Network Information Center, SRI (Sun 800-1055-01)
A 4.2BSD Interprocess Communication Primer

NAME

ip – Internet Protocol

SYNOPSIS

None; included by default with *inet*(4F).

DESCRIPTION

The Internet Protocol is designed for use in interconnected systems of packet-switched computer communication networks. It provides for transmitting blocks of data called datagrams from sources to destinations, where sources and destinations are hosts identified by fixed length addresses. It also provides for fragmentation and reassembly of long datagrams, if necessary, for transmission through "small packet" networks.

IP is specifically limited in scope. There are no mechanisms to augment end-to-end data reliability, flow control, sequencing, or other services commonly found in host-to-host protocols. IP can capitalize on the services of its supporting networks to provide various types and qualities of service.

IP is called on by host-to-host protocols, including *tcp*(4P) a reliable stream protocol, *udp*(4P) a socket-socket datagram protocol, and *nd*(4P) the network disk protocol. Other protocols may be layered on top of IP using the *raw* protocol facilities described here to receive and send datagrams with a specific IP protocol number. The IP protocol calls on local network drivers to carry the internet datagram to the next gateway or destination host.

When a datagram arrives at a UNIX host, the system performs a checksum on the header of the datagram. If this fails, or if the datagram is unreasonably short or the header length specified in the datagram is not within range, then the datagram is dropped. (Checksumming of Internet datagrams may be disabled for debugging purposes by patching the kernel variable *ipcksum* to have the value 0.)

Next the system scans the IP options of the datagram. Options allowing for source routing (see *routing*(4N)) and also the collection of time stamps as a packet follows a particular route (for network monitoring and statistics gathering purposes) are handled; other options are ignored. Processing of source routing options may result in an UNREACH *icmp*(4P) message because the source routed host is not accessible.

After processing the options, IP checks to see if the current machine is the destination for the datagram. If not, then IP attempts to forward the datagram to the proper host. Before forwarding the datagram, IP decrements the time to live field of the datagram by IPTTLDEC seconds (currently 5 from <netinet/ip.h>), and discards the datagram if its lifetime has expired, sending an ICMP TIMXCEED error packet back to the source host. Similarly if the attempt to forward the datagram fails, then ICMP messages indicating an unreachable network, datagram too large, unreachable port (datagram would have required broadcasting on the target interface, and IP does not allow directed broadcasts), lack of buffer space (reflected as a source quench), or unreachable host. Note however, in accordance with the ICMP protocol specification, ICMP messages are returned only for the first fragment of fragmented datagrams.

It is possible to disable the forwarding of datagrams by a host by patching the kernel variable *ipforwarding* to have value 0.

If a packet arrives and is destined for this machine, then IP must check to see if other fragments of the same datagram are being held. If this datagram is complete, then any previous fragments of this datagram are discarded. If this is only a fragment of a datagram, it may yield a complete set of pieces for the datagram, in which case IP constructs the complete datagram and continues processing with that. If there is yet no complete set of pieces for this datagram then we hold onto as much data as we have received (but only one copy of each data byte from the datagram) in hopes that the rest of the pieces of the fragmented datagram will arrive and we will be able to proceed. We allow IPFRAGTTL (currently 15 in <netinet/ip.h>) seconds for all the fragments of a datagram to arrive, and discard partial fragments then if the datagram has not yet been completely assembled.

When we have a complete input datagram it is passed out to the appropriate protocol's input routine: either *tcp*(4P), *udp*(4P), *nd*(4P), *icmp*(4P) or a user process through a raw IP socket as described below.

Datagrams are output by the system implemented protocols *tcp*(4P), *udp*(4P), *nd*(4P), and *icmp*(4P) as well as by packet forwarding operations and user processes through raw IP sockets. Output packets are normally subjected to routing as described in *routing*(4N); special processes such as the routing daemon *routed*(8C) occasionally use the SO_DONTROUTE socket option to cause the packets to avoid the routing tables and go directly to the network interface which has the same network number as the packet is addressed to. This is used to be able to test the ability of the hardware to transmit and receive packets even when we believe that the hardware is broken and have therefore deleted it from the routing tables.

If there is no route to a destination address or if the SO_DONTROUTE option is given and there is no interface on the network specified by the destination address, then the IP output routine returns a ENETUNREACH error. (This and the other IP output errors are reflected back to user processes through the various protocols, which individually describe how errors are reported.)

In the (hopefully normal) case where there is a suitable route or network interface, the destination address is checked to see if it specifies a broadcast (address INADDR_ANY; see *inet*(4F)); if it does, and the hardware interface does not support broadcasts, then an EADDRNOTAVAIL is returned; if the caller is not the super-user then a EACCESS error will be returned. IP also does not allow broadcast messages to be fragmented, returning a EMSGSIZE error in this case.

If the datagram passes all these tests, and is small enough to be sent in one chunk, then the system calls the output routine for the particular hardware interface to transmit the packet. The interface may give an error indication, which is reflected to IP output's caller; see the various interface's documentation for a description of the errors which they may encounter. If a datagram is to be fragmented, it may have the IP_DF (don't fragment) flag set (although currently this can happen only for forwarded datagrams). If it does, then the datagram will be rejected (and result in an ICMP error datagram). If the system runs out of buffer space in fragmenting a datagram then a ENOBUFS error will be returned.

IP provides a space of 255 protocols. The known protocols are defined in <netinet/in.h>. The ICMP, TCP, UDP and ND protocols are processed internally by the system; others may be accessed through a raw socket by doing:

s = socket(AF_INET, SOCK_RAW, IPPROTO_xxx);

Datagrams sent from this socket will have the current host's address and the specified protocol number; the raw IP driver will construct an appropriate header. When IP datagrams are received for this protocol they are queued on the raw socket where they may be read with *recvfrom;* the source IP address is reflected in the received address.

SEE ALSO

send(2), recv(2), inet(4F)
Internet Protocol, RFC791, USC-ISI (Sun 800-1063-01)

BUGS

One should be able to send and receive IP options.

Raw sockets should receive ICMP error packets relating to the protocol; currently such packets are simply discarded.

NAME

ip – Disk driver for Interphase 2180 SMD Disk Controller

SYNOPSIS

controller ipc0 at mb0 csr 0x40 priority 2
disk ip0 at ipc0 drive 0
disk ip1 at ipc0 drive 1

DESCRIPTION

Files with minor device numbers 0 through 7 refer to various portions of drive 0; minor devices 8 through 15 refer to drive 1, and so on. The standard device names begin with "ip" followed by the drive number and then a letter a-h for partitions 0-7 respectively. The character ? stands here for a drive number in the range 0-7.

The block file's access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.'

In raw I/O counts should be a multiple of 512 bytes (a disk sector). Likewise *seek* calls should specify a multiple of 512 bytes.

DISK SUPPORT

This driver handles all SMD drives, by reading a label from sector 0 of the drive which describes the disk geometry and partitioning.

The ip?a partition is normally used for the root file system on a disk, the ip?b partition as a paging area, and the ip?c partition for pack-pack copying (it normally maps the entire disk). The rest of the disk is normally the ip?h partition.

FILES

/dev/ip[0-7][a-h]	block files
/dev/rip[0-7][a-h]	raw files

SEE ALSO

dkio(4S), xy(4S)
"Interphase SMD2180 Storage Module Controller/Formatter – User's Guide" (Sun 800-0274)

DIAGNOSTICS

ip%d: SMD-2180. When booting tells the controller type.

ip%d: initialization failed. Because the controller didn't respond; perhaps another device is at the address the system expected an Interphase controller at.

ip%d: error %x reading label on head %d. Error reading drive geometry/partition table information.

ip%d: Corrupt label on head %d. The geometry/partition label checksum was incorrect.

ip%d: Misplaced label on head %d. A disk label was copied to the wrong head on the disk; shoudn't happen.

ip%d: Unsupported phys partition # %d. This indicates a bad label.

ip%d: unit not online.

ip%d%c: *cmd how (msg)* **blk %d**. A command such as read, write, or format encountered a error condition (how): either it *failed*, the unit was *restored*, or an operation was *retry*'ed. The *msg* is derived from the error number given by the controller, indicating a condition such as "drive not ready", "sector not found" or "disk write protected".

BUGS

In raw I/O *read* and *write*(2) truncate file offsets to 512-byte block boundaries, and *write* scribbles on the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, *read, write* and *lseek*(2) should always deal in 512-byte multiples.

The driver no longer supports versions of the 2181.

NAME

kb – Sun keyboard

SYNOPSIS

pseudo-device kb3

DESCRIPTION

Kb provides access to the Sun workstation keyboard translation. Definitions for alterring keyboard translation are in <sundev/kbio.h> and <sundev/kbd.h>.

The call KIOCTRANS controls the presence of keyboard translation:

```
int x;
err = ioctl(fd, KIOCTRANS, &x);
```

where if *x* is 0 the keyboard tranlation is turned off and up/down key codes are reported. Specifying *x* as 1 restores normal keyboard translations.

The call KIOCSETKEY changes a keyboard translation table entry:

```
struct  kiockey {
        int     kio_tablemask;  /* Translation table (one of: 0, CAPSMASK,
                                   SHIFTMASK, CTRLMASK, UPMASK) */
        u_char  kio_station;    /* Physical keyboard key station (0-127) */
        u_char  kio_entry;      /* Translation table station's entry */
        char    kio_string[10]; /* Value for STRING entries (null terminated) */
};
```

```
struct kiockey key;
err = ioctl(fd, KIOCSETKEY, &key);
```

Set *kio_tablemask* table's *kio_station* to *kio_entry*. Copy *kio_string* to string table if *kio_entry* is between STRING and STRING+15. This call may return EINVAL if there are invalid arguments.

The call KIOCGETKEY determines the current value of a keyboard translation table entry:

```
struct kiockey key;
err = ioctl(fd, KIOCGETKEY, &key);
```

Get *kio_tablemask* table's *kio_station* to *kio_entry*. Get *kio_string* from string table if *kio_entry* is between STRING and STRING+15. This call may return EINVAL if there are invalid arguments.

FILES

/dev/kbd

SEE ALSO

kbd(5)

NAME
lo – software loopback network interface

SYNOPSIS
pseudo-device loop

DESCRIPTION
The *loop* device is a software loopback network interface; see *if*(4N) for a general description of network interfaces.

The *loop* interface is used for performance analysis and software testing, and to provide guaranteed access to Internet protocols on machines with no local network interfaces. A typical application is the *comsat*(8C) server which accepts notification of mail delivery through a particular port on the loopback interface.

By default, the loopback interface is accessible at Internet address 127.0.0.1 (non-standard); this address may be changed with the SIOCSIFADDR ioctl.

DIAGNOSTICS
lo%d: can't handle af%d. The interface was handed a message with addresses formatted in an unsuitable address family; the packet was dropped.

SEE ALSO
if(4N), inet(4F)

BUGS
It should handle all address and protocol families. An approved network address should be reserved for this interface.

NAME

mb – Multibus

SYNOPSIS

controller mb0 at nexus ?

DESCRIPTION

The *mb* device is the driver for the Intel Multibus(R), which provides support functions to the various devices which can reside there. It vectors interrupts to the Multibus devices according to the priority level of the interrupt received and queues requests for dma when there are insufficient resources to service the request or to allow certain dma's to proceed exclusively. It also implements byte swapping to/from deficient devices.

DIAGNOSTICS

None.

SEE ALSO

ar(4S), cg(4S), ip(4S), ms(4S), oct(4S), tm(4S), vp(4S), xy(4S), zs(4S)
Intel Multibus(R) Specification, Order Number 9800683-04 (Sun 800-1057-01)

NAME

mem, kmem, mbmem, mbio – main memory and I/O space

SYNOPSIS

None; included with standard system.

DESCRIPTION

These devices are special files that map memory and bus i/o space. They may be read, written, seek'ed and (except for kmem) *mmap*(2)ed.

Mem is a special file that is an image of the physical memory of the computer. It may be used, for example, to examine (and even to patch) the system.

Kmem is a special file that is an image of the kernel virtual memory of the system.

Mbmem is a special file that is an image of the Multibus memory of the system. Multibus memory is in the range from 0 to 1 Megabyte.

Mbio is a special file that is an image of the Multibus I/O space. Multibus I/O space extends from 0 to 64K.

When reading and writing mbmem and mbio odd counts or offsets cause byte accesses and even counts and offsets cause word accesses.

DIAGNOSTICS

None.

FILES

/dev/mem
/dev/kmem
/dev/mbmem
/dev/mbio

NAME

mouse – Sun mouse

SYNOPSIS

pseudo-device ms3

DESCRIPTION

The *mouse* interface provides access to the Sun Workstation mouse.

The mouse incorporates a microprocessor which generates a byte-stream protocol encoding mouse motions.

Each mouse sample in the byte stream consists of three bytes: the first byte gives the button state with value 0x87|~*but*, where *but* is the low three bits giving the mouse buttons, where a 0 (zero) bit means that a button is pressed, and a 1 (one) bit means a button is not pressed. Thus if the left button is down the value of this sample is 0x83, while if the right button is down the byte is 0x86.

The next two bytes of each sample give the *x* and *y delta's of this* sample as signed bytes. The mouse uses a lower-left coordinate system, so moves to the right on the screen yield positive *x* values and moves down the screen yield negative *y* values.

The beginning of a sample is identifiable because the delta's are constrained to not have values in the range 0x80-0x87.

FILES

/dev/mouse

SEE ALSO

win(4S)
Mouse System Mouse Manual (Sun 800-0419)
User's Guide for the Sun Workstation Mouse Subsystem (Sun 800-0402)

NAME

mti – Systech MTI-800/1600 multi-terminal interface

SYNOPSIS

device mti0 at mb0 csr 0x620 flags 0xffff priority 4

DESCRIPTION

The Systech MTI card provides 8 (MTI-800) or 16 (MTI-1600) serial communication lines with modem control. Each line behaves as described in *tty*(4). Input and output for each line may independently be set to run at any of 16 speeds; see *tty*(4) for the encoding.

Bit *i* of flags may be specified to say that a line is not properly connected, and that the line *i* should be treated as hard-wired with carrier always present. Thus specifying "flags 0x0004" in the specification of mti0 would cause line tty02 to be treated in this way.

To allow a single tty line to be connected to a modem and used for both incoming and outgoing calls, a special feature, controlled by the minor device number, has been added. Minor device numbers in the range 0 – 127 correspond directly to the normal tty lines and are named *tty**. Minor device numbers in the range 128 – 256 correspond to the same physical lines as those above (i.e. the same line as the minor device number minus 128) and are (conventionally) named *cua**. The *cua* lines are special in that they can be opened even when there is no carrier on the line. Once a *cua* line is opened, the corresponding tty line can not be opened until the *cua* line is closed. Also, if the *tty* line has been opened successfully (usually only when carrier is recognized on the modem) the corresponding *cua* line can not be opened. This allows a modem to be attached to */dev/tty00* (usually renamed to */dev/ttyd0*) and used for dialin (be enabling the line for login in */etc/ttys*) and also used for dialout (by *tip*(1C) or *uucp*(1C)) as */dev/cua0* when no one is logged in on the line. Note that the bit in the flags word in the config file (see above) must be zero for this line.

WIRING

The Systech requires the CTS modem control signal to operate. If the device does not supply CTS then RTS should be jumpered to CTS at the distribution panel. Also, the CD (carrier detect) line does not work properly. When connecting a modem, the modem's CD line should be wired to DSR, which the software will treat as carrier detect.

FILES

/dev/tty0[0-9a-f]	hardwired tty lines
/dev/ttyd[0-9a-f]	dialin tty lines
/dev/cua[0-9a-f]	dialout tty lines

SEE ALSO

tty(4), zs(4S)

DIAGNOSTICS

Most of these diagnostics "should never happen" and their occurrence usually indicates problems elsewhere in the system.

mti%d,%d: silo overflow. More than 512 characters have been received by the mti hardware without being read by the software. Extremely unlikely to occur.

mti%d: error %x. The mti returned the indicated error code. See the mti manual.

mti%d: DMA output error. The mti encountered an error while trying to do DMA output.

mti%d: impossible response %x. The mti returned of flags may be specified to say that a line is not From marty@ufo Sat Feb 25 16:04:52 1984

NAME

mtio – UNIX magnetic tape interface

SYNOPSIS

#include <sys/ioctl.h>
#include <sys/mtio.h>

DESCRIPTION

The files *mt0, ..., mt15* refer to the UNIX magtape drives, which read and write magnetic tape in 2048 byte blocks. (The 2048 is actually BLKDEV_IOSIZE in <sys/param.h>.) The following description applies to any of the transport/controller pairs. The files *mt0, ..., mt3* and *mt8, ..., mt11* are rewound when closed; the others are not. When a file open for writing is closed, two end-of-files are written. If the tape is not to be rewound it is positioned with the head between the two tapemarks.

The *mt* files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the 'raw' interface is appropriate. The associated files are named *rmt0, ..., rmt15,* but the same minor-device considerations as for the regular files still apply. A number of other ioctl operations are available on raw magnetic tape. The following definitions are from <sys/mtio.h>:

```
/*
 * Structures and definitions for mag tape io control commands
 */

/* structure for MTIOCTOP - mag tape op command */
struct  mtop  {
        short   mt_op;          /* operations defined below */
        daddr_t mt_count;       /* how many of them */
};

/* operations */
#define MTWEOF      0       /* write an end-of-file record */
#define MTFSF       1       /* forward space file */
#define MTBSF       2       /* backward space file */
#define MTFSR       3       /* forward space record */
#define MTBSR       4       /* backward space record */
#define MTREW       5       /* rewind */
#define MTOFFL      6       /* rewind and put the drive offline */
#define MTNOP       7       /* no operation, sets status only */

/* structure for MTIOCGET - mag tape get status command */

struct  mtget  {
        short   mt_type;        /* type of magtape device */
/* the following two registers are grossly device dependent */
        short   mt_dsreg;       /* "drive status" register */
        short   mt_erreg;       /* "error" register */
/* end device-dependent registers */
        short   mt_resid;       /* residual count */
/* the following two are not yet implemented */
        daddr_t mt_fileno;      /* file number of current position */
        daddr_t mt_blkno;       /* block number of current position */
/* end not yet implemented */
};
```

```
/*
 * Constants for mt_type byte
 */
#define MT_ISTS      0x01          /* vax: unibus ts-11 */
#define MT_ISHT      0x02          /* vax: massbus tu77, etc */
#define MT_ISTM      0x03          /* vax: unibus tm-11 */
#define MT_ISMT      0x04          /* vax: massbus tu78 */
#define MT_ISUT      0x05          /* vax: unibus gcr */
#define MT_ISCPC     0x06          /* sun: Multibus tapemaster */
#define MT_ISAR      0x07          /* sun: Multibus archive */

/* mag tape io control commands */
#define MTIOCTOP     _IOW(m, 1, struct mtop)      /* do a mag tape op */
#define MTIOCGET     _IOR(m, 2, struct mtget)     /* get tape status */

#ifndef KERNEL
#define DEFTAPE      "/dev/rmt12"
#endif
```

Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, provided it is no greater than the buffer size. In raw tape I/O seeks are ignored. A zero byte count is returned when a tape mark is read, but another read will fetch the first record of the new tape file.

FILES

/dev/mt?
/dev/rmt?
/dev/rar?

SEE ALSO

mt(1), tar(1), ar(4S), tm(4S)

NAME

nd – network disk driver

SYNOPSIS

pseudo-device nd

DESCRIPTION

The network disk device, */dev/nd**, allows a *client* workstation to perform disk IO operations on a *server* system, over the network. To the client system, this device looks like any normal disk driver: it allows read/write operations at a given block number and byte count. Note that this provides a network *disk block* access service rather than a network *file* access service.

Typically the client system will have no disks at all. In this case */dev/nd0* contains the client's root file system (including /usr files), and *nd1* is used as a paging area. Client access to these devices is converted to *net disk protocol* requests and sent to the server system over the network. The server receives the request, performs the actual disk IO, and sends a response back to the client.

The server contains a table which lists the net address of each of his clients and the server disk partition which corresponds to each client unit number (nd0,1,...). This table resides in the server kernel in a structure owned by the nd device. The table is initialized by running the program */etc/nd* with text file */etc/nd.local* as its input. */etc/nd* then issues *ioctl*(2) functions to load the table into the kernel.

In addition to the read/write units */dev/nd**, there are *public* read-only units which are named */dev/ndp**. The correspondence to server partitions is specified by the */etc/nd.local* text file, in a similar manner to the private partitions. The public units can be used to provide shared access to binaries or libraries (/bin, /usr/bin, /usr/ucb, /usr/lib) so that each diskless client does not have to waste space in his private partitions for these files. This is done by providing a public file system at the server (*/dev/ndp0*) which is mounted on '/pub' of each diskless client. The clients then use symbolic links to read the public files: /bin -> /pub/bin, /usr/ucb -> /pub/usr/ucb. One requirement in this case is that the server (who has read/write access to this file system) should not perform write activity with any public filesystem. This is because each client is locally cacheing blocks.

One last type of unit is provided for use by the server. These are called *local* units and are named */dev/ndl**. The Sun physical disk sector 0 label only provides a limited number of partitions per physical disk (eight). Since this number is small and these partitions have somewhat fixed meanings, the nd driver itself has a *subpartitioning* capability built-in. This allows the large server physical disk partition (e.g. */dev/xy0g*) to be broken up into any number of diskless client partitions. Of course on the client side these would be referenced as */dev/nd0,1,...* ; but the server needs to reference these client partitions from time to time, to do *mkfs*(8) and *fsck*(8) for example. The */dev/ndl** entries allow the server 'local' access to his subpartitions without causing any net activity. The actual local unit number to client unit number correspondence is again recorded in the */etc/nd.local* text file.

The nd device driver is the same on both the client and server sides. There are no user level processes associated with either side, thus the latency and transfer rates are close to maximal.

The minor device and ioctl encoding used is given in file *<sun/ndio.h>*. The low six bits of the minor number are the unit number. The 0x40 bit indicates a *public* unit; the 0x80 bit indicates a *local* unit.

INITIALIZATION

No special initialization is required on the client side; he finds the server by broadcasting the initial request. Upon getting a response, he locks onto that server address.

At the server, the *nd*(8c) command initializes the network disk service by issuing ioctl's to the kernel.

ERRORS

Generally physical disk IO errors detected at the server are returned to the client for action. If the server is down or unaccessable, the client will see the console message *file server not responding: still trying.* The client continues (forever) making his request until he gets positive acknowledgement from the server. This means the server can crash or power down and come back up without any special action required of the user at the client machine. It also means the process performing the IO to nd will block, insensitive to signals, since the process is sleeping inside the kernel at PRIBIO.

PROTOCOL AND DRIVER INTERNALS

The protocol packet is defined in *<sun/ndio.h>* and also included below:

```
/*
 * 'nd' protocol packet format.
 */
struct ndpack {
        struct  ip np_ip;       /* ip header, proto IPPROTO_ND */
        u_char np_op;           /* operation code, see below */
        u_char np_min;          /* minor device */
        char    np_error;       /* b_error */
        char    np_ver;         /* version number */
        long    np_seq;         /* sequence number */
        long    np_blkno;       /* b_blkno, disk block number */
        long    np_bcount;      /* b_bcount, byte count */
        long    np_resid;       /* b_resid, residual byte count */
        long    np_caddr;       /* current byte offset of this packet */
        long    np_ccount;      /* current byte count of this packet */
};                              /* data follows */

/* np_op operation codes. */
#define NDOPREAD        1       /* read */
#define NDOPWRITE       2       /* write */
#define NDOPERROR       3       /* error */
#define NDOPCODE        7       /* op code mask */
#define NDOPWAIT        010     /* waiting for DONE or next request */
#define NDOPDONE        020     /* operation done */

/* misc protocol defines. */
#define NDMAXDATA       1024    /* max data per packet  */
#define NDMAXIO         63*1024         /* max np_bcount */
```

IP datagrams were chosen instead of UDP datagrams because only the IP header is checksummed, not the entire packet as in UDP. Also the kernel level interface to the IP layer is simpler. The *min, blkno,* and *bcount* fields are copied directly from the client's strategy request. The sequence number field *seq* is incremented on each new client request and is matched with incoming server responses. The server essentially echos the request header in his responses, altering certain fields. The *caddr* and *ccount* fields show the current byte address and count of the data in this packet, or the data expected to be sent by the other side.

The protocol is very simple and driven entirely from the client side. As soon as the client ndstrategy routine is called, the request is sent to the server; this allows disk sorting to occur at the server as soon as possible. Transactions which send data (client writes on the client side, client reads on the server side) can only send a set number of packets of NDMAXDATA bytes each, before waiting for an acknowledgement. The defaults are currently set at 6 packets of 1K bytes each; the NDIOCETHER ioctl allows setting this value on the server side. This allows the normal 4K byte case to occur with just one 'transaction'. The NDOPWAIT bit is set in the *op* field by the sender to indicate he will send no more until acknowledged (or requested) by the other side. The NDOPDONE bit is set by the server side to indicate the request operation has completed; for both the read and write cases this means the requested disk IO has actually occured.

Requests received by the server are entered on an active list which is timed out and discarded if not completed within NDXTIMER seconds. Requests received by the server allocate a *bcount* size buffer to minimize buffer copying. Contiguous DMA disk IO thus occurs in the same size chunks it would if requested from a local physical disk.

BOOTSTRAP

The Sun workstation has PROM code to perform a net boot using this driver. Usually, the boot files are obtained from public device 0 (/dev/ndp0) on the server with which the client is registered; this allows multiple servers to exist on the same net (even running different releases of kernel and boot software). If the station you are booting is not registered on any of the servers, you will have to specify the hex Internet host number of the server in the boot command string (e.g.): 'bec(0,5,0)vmunix'.

This booting performs exactly the same steps involved in a real disk boot which are:

1) user types 'b' to PROM monitor.

2) PROM loads blocks 1 thru 15 of /dev/ndp0 (bootpr).

3) bootnd loads '/boot'.

4) /boot loads '/vmunix'.

SEE ALSO

ioctl(2), nd(8C)

BUGS

The operations described in *dkio*(4) are not supported.

The local host's disk buffer cache is not used by network disk access. This means that if either a local host or a remote host is writing, the changes will be visible at random based on the cache hit frequency on the local host. If both the local and remote hosts are writing to the same filesystem, one machine's changes can be randomly lost, based again on cache hit and deferred write timings.

If an R/O remote file system is mounted R/W by mistake, it is impossible to umount it.

NAME

null – data sink

SYNOPSIS

None; included with standard system.

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return an end-of-file indication.

FILES

/dev/null

NAME

oct – Central Data octal serial card

SYNOPSIS

device oct0 at mb0 csr 0x520 flags 0xff priority 4

DESCRIPTION

The Central Data card provides 8 serial communication lines with modem control. Each line behaves as described in *tty*(4). Input and output for each line may independently be set to run at any of 16 speeds; see *tty*(4) for the encoding.

Bit *i* of flags may be specified to say that a line is not properly connected, and that the line *i* should be treated as hard-wired with carrier always present. Thus specifying "flags 0x0004" in the specification of oct0 would cause line ttym2 to be treated in this way.

FILES

/dev/tty[mno][0-9a-f]
/dev/ttyd[0-9a-f]

SEE ALSO

tty(4), zs(4S)
Hardware Reference Manual; Octal Serial Interface; Central Data Corporation (Sun 800-0418)

DIAGNOSTICS

None.

BUGS

Input data overruns are silently ignored.

This interrupt-per-character, non-buffered device is expensive in terms of system overhead.

This driver is not supported.

NAME

pty – pseudo terminal driver

SYNOPSIS

pseudo-device pty

DESCRIPTION

The *pty* driver provides support for a pair of devices collectively known as a *pseudo-terminal.* The two devices comprising a pseudo-terminal are known as a *master* and a *slave.* The slave device provides an interface identical to that described in *tty*(4), but instead of having a hardware interface such as the Zilog chip and associated hardware used by *zs*(4S) supporting the terminal functions, the functions of the terminal are implemented by another process manipulating the master side of the pseudo-terminal.

The master and the slave sides of the pseudo-terminal are tightly connected. Any data written on the master device is given to the slave device as input, as though it had been received from a hardware interface. Any data written on the slave terminal can be read from the master device (rather than being transmitted from a UART).

In configuring, if no optional "count" is given in the specification, 16 pseudo terminal pairs are configured.

A few special ioctl's are provided on the control-side devices of pseudo-terminals to provide the functionality needed by applications programs to emulate real hardware interfaces:

TIOCSTOP
: Stops output to a terminal (that is, like typing ^S). Takes no parameter.

TIOCSTART
: Restarts output (stopped by TIOCSTOP or by typing ^Q). Takes no parameter.

There are also two independent modes which can be used by applications programs:

TIOCPKT
: Enable/disable *packet* mode. Packet mode is enabled by specifying (by reference) a nonzero parameter and disabled by specifying (by reference) a zero parameter. When applied to the master side of a pseudo terminal, each subsequent *read* from the terminal will return data written on the slave part of the pseudo terminal preceded by a zero byte (symbolically defined as TIOCPKT_DATA), or a single byte reflecting control status information. In the latter case, the byte is an inclusive-or of zero or more of the bits:

  TIOCPKT_FLUSHREAD
  : whenever the read queue for the terminal is flushed.

  TIOCPKT_FLUSHWRITE
  : whenever the write queue for the terminal is flushed.

  TIOCPKT_STOP
  : whenever output to the terminal is stopped a la ^S.

  TIOCPKT_START
  : whenever output to the terminal is restarted.

  TIOCPKT_DOSTOP
  : whenever *t_stopc* is ^S and *t_startc* is ^Q.

  TIOCPKT_NOSTOP
  : whenever the start and stop characters are not ^S/^Q.

  This mode is used by *rlogin*(1C) and *rlogind*(8C) to implement a remote-echoed, locally ^S/^Q flow-controlled remote login with proper back-flushing of output when interrupts occur; it can be used by other similar programs.

TIOCREMOTE

A mode for the master half of a pseudo terminal, independent of TIOCPKT. This mode causes input to the pseudo terminal to be flow controlled and not input edited (regardless of the terminal mode). Each write to the control terminal produces a record boundary for the process reading the terminal. In normal usage, a write of data is like the data typed as a line on the terminal; a write of 0 bytes is like typing an end-of-file character. TIOCREMOTE can be used when doing remote line editing in a window manager, or whenever flow controlled input is required.

FILES

/dev/pty[p-r][0-9a-f]	master pseudo terminals
/dev/tty[p-r][0-9a-f]	slave pseudo terminals

BUGS

It is apparently not possible to send an EOT by writing zero bytes in TIOCREMOTE mode.

NAME

routing – system supporting for local network packet routing

DESCRIPTION

The network facilities provided general packet routing, leaving routing table maintenance to applications processes.

A simple set of data structures comprise a "routing table" used in selecting the appropriate network interface when transmitting packets. This table contains a single entry for each route to a specific network or host. A user process, the routing daemon, maintains this data base with the aid of two socket specific *ioctl*(2) commands, SIOCADDRT and SIOCDELRT. The commands allow the addition and deletion of a single routing table entry, respectively. Routing table manipulations may only be carried out by super-user.

A routing table entry has the following form, as defined in <*net/route.h*>;

```
struct rtentry {
        u_long  rt_hash;
        struct  sockaddr rt_dst;
        struct  sockaddr rt_gateway;
        short   rt_flags;
        short   rt_refcnt;
        u_long  rt_use;
        struct  ifnet *rt_ifp;
};
```

with *rt_flags* defined from,

```
#define RTF_UP          0x1     /* route usable */
#define RTF_GATEWAY     0x2     /* destination is a gateway */
#define RTF_HOST        0x4     /* host entry (net otherwise) */
```

Routing table entries come in three flavors: for a specific host, for all hosts on a specific network, for any destination not matched by entries of the first two types (a wildcard route). When the system is booted, each network interface autoconfigured installs a routing table entry when it wishes to have packets sent through it. Normally the interface specifies the route through it is a "direct" connection to the destination host or network. If the route is direct, the transport layer of a protocol family usually requests the packet be sent to the same host specified in the packet. Otherwise, the interface may be requested to address the packet to an entity different from the eventual recipient (i.e. the packet is forwarded).

Routing table entries installed by a user process may not specify the hash, reference count, use, or interface fields; these are filled in by the routing routines. If a route is in use when it is deleted (*rt_refcnt* is non-zero), the resources associated with it will not be reclaimed until further references to it are released.

The routing code returns EEXIST if requested to duplicate an existing entry, ESRCH if requested to delete a non-existant entry, or ENOBUFS if insufficient resources were available to install a new route.

User processes read the routing tables through the */dev/kmem* device.

The *rt_use* field contains the number of packets sent along the route. This value is used to select among multiple routes to the same destination. When multiple routes to the same destination exist, the least used route is selected.

A wildcard routing entry is specified with a zero destination address value. Wildcard routes are used only when the system fails to find a route to the destination host and network. The combination of wildcard routes and routing redirects can provide an economical mechanism for routing traffic.

SEE ALSO

route(8C), routed(8C)

NAME

sd – Disk driver for Adaptec ST-506 Disk Controllers

SYNOPSIS

controller sc0 at mb0 csr 0x80000 priority 2
disk sd0 at sc0 drive 0 flags 0
disk sd1 at sc0 drive 1 flags 0

DESCRIPTION

Files with minor device numbers 0 through 7 refer to various portions of drive 0. The standard device names begin with "sd" followed by the drive number and then a letter a-h for partitions 0-7 respectively. The character ? stands here for a drive number in the range 0-7.

The block file's access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.'

In raw I/O counts should be a multiple of 512 bytes (a disk sector). Likewise *seek* calls should specify a multiple of 512 bytes.

DISK SUPPORT

This driver handles all ST-506 drives, by reading a label from sector 0 of the drive which describes the disk geometry and partitioning.

The sd?a partition is normally used for the root file system on a disk, the sd?b partition as a paging area, and the sd?c partition for pack-pack copying (it normally maps the entire disk). The rest of the disk is normally the sd?h partition.

FILES

/dev/sd[0-7][a-h]	block files
/dev/rsd[0-7][a-h]	raw files

SEE ALSO

dkio(4S)
Adaptec ACB 4000 and 5000 Series Disk Controllers OEM Manual

DIAGNOSTICS

sd%d%c: *cmd how (msg)* **blk %d**. A command such as read or write encountered a error condition (how): either it *failed*, the unit was *restored*, or an operation was *retry*'ed. The *msg* is derived from the error number given by the controller, indicating a condition such as "drive not ready" or "sector not found".

BUGS

In raw I/O *read* and *write*(2) truncate file offsets to 512-byte block boundaries, and *write* scribbles on the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, *read, write* and *lseek*(2) should always deal in 512-byte multiples.

NAME

st – Driver for Sysgen SC 4000 (Archive) Tape Controller

SYNOPSIS

controller sc0 at mb0 csr 0x80000 priority 2
tape st0 at sc0 drive 32 flags 1

DESCRIPTION

The Sysgen tape controller is a SCSI bus interface to an Archive streaming tape drive. It provides a standard tape interface to the device, see *mtio*(4), with some deficiencies listed under BUGS below.

FILES

/dev/rst0
/dev/nrst0 non-rewinding

SEE ALSO

mtio(4), tm(4S)
Sysgen SC4000 Intelligent Tape Controller Product Specification
Archive Intelligent Tape Drive Theory of Operation, Archive Corporation (Sun 8000-1058-01)
Archive Product Manual (Sidewinder 1/4" Streaming Cartridge Tape Drive) (Sun 800-0628-01)

DIAGNOSTICS

st%d: tape not online.

st%d: no cartridge in drive.

st%d: cartridge is write protected.

BUGS

The tape cannot reverse direction so BSF and BSR are not available.

Disk I/O over the SCSI bus will be mostly blocked out when the tape is in use. This is because the controller does not free the bus while the tape is in motion (even during rewind).

When using the raw device, the number of bytes in any given transfer must be a multiple of 512 bytes. If it is not, the device driver returns an error.

NAME

tcp – Internet Transmission Control Protocol

SYNOPSIS

None; comes automatically with *inet*(4F).

DESCRIPTION

TCP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarch of protocols which support multi-network applications. TCP provides for reliable inter-process communication between pairs of processes in host computers attached to distinct but interconnected computer communication networks. Very few assumptions are made as to the reliability of the communication protocols below TCP layer. TCP assumes it can obtain a simple, potentially unreliable datagram service from the lower level protocols. In principle, TCP should be able to operate above a wide spectrum of communication systems ranging from hard-wired connections to packet-switched or circuit switched networks.

TCP fits into a layered protocol architecture just above the basic Internet Protocol (IP) described in *ip*(4P) which provides a way for TCP to send and receive variable-length segments of information enclosed in Internet datagram "envelopes." The Internet datagram provides a means for addressing source and destination TCPs in different networks, deals with any fragmentation or reassembly of the TCP segments required to achieve transport and delivery through multiple netwokrs and interconnecting gateways, and has the ability to carry information on the precedence, security classification and compartmentalization of the TCP segments (although this is not currently implemented under UNIX.)

An application process interfaces to TCP through the *socket*(2) abstraction and the related calles *bind*(2), *listen*(2), *accept*(2), *connect*(2), *send*(2) and *recv*(2). The primary purpose of TCP is to provide a reliable bidirectional virtual circuit service between pairs of processes. In general, the TCP's decide when to block and forward data at their own convenience. In the UNIX implementation, it is assumed that any buffering of data is done at the user level, and the TCP's transmit available data as soon as possible to their remote peer. They do this and always set the PUSH bit indicating that the transferred data should be made available to the user process at the remote end as soon as practicable.

To provide reliable data TCP must recover from data that is damaged, lost, duplicated, or delivered out of order by the underlying internet communications system. This is achieved by assigning a sequence number to each byte of data transmitted and requiring a positive acknowledgement from the receiving TCP. If the ACK is not received within an (adaptively determined) timeout interval the data is retransmitted. At the receiver, the sequence numbers are used to correctly order segments that may be received out of order and to eliminate duplicates. Damage is handled by adding a checksum to each segment transmitted, checking it at the receiver, and discarding damaged segments. As long as the TCP's continue to function properly and the internet system does not become completely partitioned, no tranmission errors will affect the correct delivery of data, as TCP recovers from communications errors.

TCP provides flow control over the transmitted data. The receiving TCP is allowed to specify the amount of data which may be sent by the sender, by returning a *window* with every acknowledgement indicating a range of acceptable sequence numbers beyond the last segment successfully received. The window indicates an allowed number of bytes that the sender may transmit before receiving further permission.

TCP extends the standard 32-bit Internet host addresses with a 16-bit port number space; the combined addresses are available at the UNIX process level in the standard *sockaddr_in* format described in *inet*(4F).

Sockets utilizing the tcp protocol are either "active" or "passive". Active sockets initiate connections to passive sockets. By default TCP sockets are created active; to create a passive socket the *listen*(2) system call must be used after binding the socket to an address with the *bind*(2) system call. Only passive sockets may use the *accept*(2) call to accept incoming connections. Only active sockets may use the *connect*(2) call to initiate connections.

Passive sockets may "underspecify" their location to match incoming connection requests from multiple networks. This technique, termed "wildcard addressing", allows a single server to provide service to clients on multiple networks. To create a socket which listens on all networks, the Internet address INADDR_ANY must be bound. The TCP port may still be specified at this time; if the port is not specified the system will assign one. Once a connection has been established the socket's address is fixed by the peer entity's location. The address assigned the socket is the address associated with the network interface through which packets are being transmitted and received. Normally this address corresponds to the peer entity's network. See *inet*(4F) for a complete description of addressing in the Internet family.

A TCP connection is created at the server end by doing a *socket*(2), a *bind*(2) to establish the address of the socket, a *listen*(2) to cause connection queueing, and then an *accept*(2) which returns the descriptor for the socket. A client connects to the server by doing a *socket*(2) and then a *connect*(2). Data may then be sent from server to client and back using *read*(2) and *write*(2).

TCP implements a very weak out-of-band mechanism, which may be invoked using the out-of-band provisions of *send*(2). This mechanism allows setting an urgent pointer in the data stream; it is reflected to the TCP user by making the byte after the urgent pointer available as out-of-band data and providing a SIOCATMARK ioctl which returns an integer indicating whether the stream is at the urgent mark. The system never returns data across the urgent mark in a single read. Thus when a SIGURG signal is received indicating the presence of out-of-band data and the out-of-band data indicates that the data to the mark should be flushed (as in remote terminal processing) it suffices to loop checking whether you are at the out-of-band mark, and reading data while you are not at the mark.

SEE ALSO

inet(4F), ip(4P)

BUGS

It should be possible to send and receive TCP options.

The system always tries to negotiates the maximum TCP segment size to be 1024 bytes. This can result in poor performance if an intervening network performs excessive fragmentation.

SIOCSHIWAT and SIOCGHIWAT ioctl's to set and get the high water mark for the socket queue, and so that it can be changed from 2048 bytes to be larger or smaller, have been defined (in <sys/ioctl.h>) but not implemented.

NAME

tm – tapemaster 1/2 inch tape drive

SYNOPSIS

controller tm0 at mb0 csr 0xa0 priority 3
tape mt0 at tm0 drive 0 flags 1

DESCRIPTION

The Tapemaster tape controller controls Pertec-interface 1/2" tape drives such as the CDC Keystone, providing a standard tape interface to the device, see *mtio*(4).

SEE ALSO

mt(1), tar(1), ar(4S)
CPC Tapemaster Product Specification (Sun 800-0620-01)
CPC Tapemaster Application Note (Sun 800-0622-01)
CDC Streaming Tape Unit 9218X Reference Manual (Sun 800-0623-01)

DIAGNOSTICS

tm%d: no response from ctlr.

tm%d: error %d during config.

mt%d: not online.

mt%d: no write ring.

tmgo: gate wasn't open. Controller lost synch.

tmintr: can't clear interrupts.

tm%d: stray interrupts.

mt%d: hard error bn=%d er=%x.

mt%d: lost interrupt.

BUGS

The Tapemaster controller does not provide for byte-swapping and the resultant system overhead prevents streaming transports from streaming.

If a non-data error is encountered on non-raw tape, it refuses to do anything more until closed.

The system should remember which controlling terminal has the tape drive open and write error messages to that terminal rather than on the console.

NAME

tty – general terminal interface

SYNOPSIS

None; included by default.

DESCRIPTION

This section describes both a particular special file **/dev/tty** and the terminal drivers used for conversational computing by serial interfaces such as *oct*(4S), *zs*(4S), as well as *cons*(4S) and *pty*(4).

Line disciplines.

The system provides different *line disciplines* for controlling communications lines. In this version of the system there are three disciplines available:

old The old (standard) terminal driver. This is used when using the standard shell *sh*(1) and for compatibility with version 7 UNIX systems.

new A newer terminal driver, with features for job control; this must be used when using *csh*(1).

net A line discipline used for networking and loading data into the system over communications lines. It allows high speed input at very low overhead, and is described in *bk*(4).

Line discipline switching is accomplished with the TIOCSETD *ioctl:*

```
int ldisc = LDISC; ioctl(f, TIOCSETD, &ldisc);
```

where LDISC is OTTYDISC for the standard tty driver, NTTYDISC for the new driver and NETLDISC for the networking discipline. The standard (currently old) tty driver is discipline 0 by convention. The current line discipline can be obtained with the TIOCGETD ioctl. Pending input is discarded when the line discipline is changed.

All of the low-speed asynchronous communications ports can use any of the available line disciplines, no matter what hardware is involved. The remainder of this section discusses the "old" and "new" disciplines.

The control terminal.

When a terminal file is opened, it causes the process to wait until a connection is established. In practice, user programs seldom open these files; they are opened by *init*(8) and become a user's standard input and output file.

If a process which has no control terminal opens a terminal file, then that terminal file becomes the control terminal for that process. The control terminal is thereafter inherited by a child process, during a *fork*(2), even if the control terminal is closed.

The file **/dev/tty** is, in each process, a synonym for a *control terminal* associated with that process. It is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand a file name for output, when typed output is desired and it is tiresome to find out which terminal is currently in use.

A process can remove the association it has with its controlling terminal by opening the file **/dev/tty** and issuing a

```
ioctl(f, TIOCNOTTY, 0);
```

This is often desirable in server processes.

Process groups.

Command processors such as *csh*(1) can arbitrate the terminal between different *jobs* by placing related jobs in a single process group and associating this process group with the terminal. A terminal's associated process group may be set using the TIOCSPGRP *ioctl*(2):

ioctl(fildes, TIOCSPGRP, &pgrp)

or examined using TIOCGPGRP, returning the current process group in *pgrp*. The new terminal driver aids in this arbitration by restricting access to the terminal by processes which are not in the current process group; see **Job access control** below.

Modes.

The terminal drivers have three major modes, characterized by the amount of processing on the input and output characters:

cooked The normal mode. In this mode lines of input are collected and input editing is done. The edited line is made available when it is completed by a newline or when the *t_brkc* character, normally an EOT (control-D, hereafter ^D), is entered. A carriage return is usually made synonymous with newline in this mode, and replaced with a newline whenever it is typed. All driver functions (input editing, interrupt generation, output processing such as delay generation and tab expansion, etc.) are available in this mode.

CBREAK This mode eliminates the character, word, and line editing input facilities, making the input character available to the user program as it is typed. Flow control, literal-next and interrupt processing are still done in this mode. Output processing is done.

RAW This mode eliminates all input processing and makes all input characters available as they are typed; no output processing is done either.

The style of input processing can also be very different when the terminal is put in non-blocking i/o mode; see the FNDELAY flag as described in *fcntl*(2). In this case a *read*(2) from the control terminal will never block, but rather return an error indication (EWOULDBLOCK) if there is no input available.

A process may also request a SIGIO signal be sent it whenever input is present. To enable this mode the FASYNC flag should be set using *fcntl*(2).

Input editing.

A UNIX terminal ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely choked, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently this limit is 256 characters. In the old terminal driver all the saved characters are thrown away when the limit is reached, without notice; the new driver simply refuses to accept any further input, and rings the terminal bell.

Input characters are normally accepted in either even or odd parity with the parity bit being stripped off before the character is given to the program. By clearing either the EVEN or ODD bit in the flags word it is possible to have input characters with that parity discarded (see the **Summary** below.)

In all of the line disciplines, it is possible to simulate terminal input using the TIOCSTI ioctl, which takes, as its third argument, the address of a character. The system pretends that this character was typed on the argument terminal, which must be the control terminal except for the super-user (this call is not in standard version 7 UNIX).

Input characters are normally echoed by putting them in an output queue as they arrive. This may be disabled by clearing the ECHO bit in the flags word using the *stty*(3C) call or the TIOCSETN or TIOCSETP ioctls (see the **Summary** below).

In cooked mode, terminal input is processed in units of lines. A program attempting to read will normally be suspended until an entire line has been received (but see the description of SIGTTIN in **Modes** above and FIONREAD in **Summary** below.) No matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, line editing is normally done, with the DELETE character logically erasing the last character typed and a ^U (control-U) logically erasing the entire current input line. These characters never erase beyond the beginning of the current input line or an ^D. These characters may be entered literally by preceding them with '\'; in the old teletype driver both the '\' and the character entered literally will appear on the screen; in the new driver the '\' will normally disappear.

The drivers normally treat either a carriage return or a newline character as terminating an input line, replacing the return with a newline and echoing a return and a line feed. If the CRMOD bit is cleared in the local mode word then the processing for carriage return is disabled, and it is simply echoed as a return, and does not terminate cooked mode input.

In the new driver there is a literal-next character ^V which can be typed in both cooked and CBREAK mode preceding **any** character to prevent its special meaning. This is to be preferred to the use of '\' escaping erase and kill characters, but '\' is (at least temporarily) retained with its old function in the new driver for historical reasons.

The new terminal driver also provides two other editing characters in normal mode. The word-erase character, normally ^W, erases the preceding word, but not any spaces before it. For the purposes of ^W, a word is defined as a sequence of non-blank characters, with tabs counted as blanks. Finally, the reprint character, normally ^R, retypes the pending input beginning on a new line. Retyping occurs automatically in cooked mode if characters which would normally be erased from the screen are fouled by program output.

Input echoing and redisplay

In the old terminal driver, nothing special occurs when an erase character is typed; the erase character is simply echoed. When a kill character is typed it is echoed followed by a new-line (even if the character is not killing the line, because it was preceded by a '\'!.)

The new terminal driver has several modes for handling the echoing of terminal input, controlled by bits in a local mode word.

Hardcopy terminals. When a hardcopy terminal is in use, the LPRTERA bit is normally set in the local mode word. Characters which are logically erased are then printed out backwards preceded by '\' and followed by '/' in this mode.

Crt terminals. When a crt terminal is in use, the LCRTBS bit is normally set in the local mode word. The terminal driver then echoes the proper number of backspace characters when input is erased to reposition the cursor. If the input has become fouled due to interspersed asynchronous output, the input is automatically retyped.

Erasing characters from a crt. When a crt terminal is in use, the LCRTERA bit may be set to cause input to be erased from the screen with a "backspace-space-backspace" sequence when character or word deleting sequences are used. A LCRTKIL bit may be set as well, causing the input to be erased in this manner on line kill sequences as well.

Echoing of control characters. If the LCTLECH bit is set in the local state word, then non-printing (control) characters are normally echoed as ^X (for some X) rather than being echoed unmodified; delete is echoed as ^?.

The normal modes for using the new terminal driver on crt terminals are speed dependent. At speeds less than 1200 baud, the LCRTERA and LCRTKILL processing is painfully slow, so *stty*(1) normally just sets LCRTBS and LCTLECH; at speeds of 1200 baud or greater all of these bits are normally set. The *stty*(1) command summarizes these option settings and the use of the new terminal driver as "newcrt."

Output processing.

When one or more characters are written, they are actually transmitted to the terminal as soon as previously-written characters have finished typing. (As noted above, input characters are normally echoed by putting them in the output queue as they arrive.) When a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold the program is resumed. Even parity is normally generated on output. The EOT character is not transmitted in cooked mode to prevent terminals that respond to it from hanging up; programs using raw or cbreak mode should be careful.

The terminal drivers provide necessary processing for cooked and CBREAK mode output including delay generation for certain special characters and parity generation. Delays are available after backspaces ^H, form feeds ^L, carriage returns ^M, tabs ^I and newlines ^J. The driver will also optionally expand tabs into spaces, where the tab stops are assumed to be set every eight columns. These functions are controlled by bits in the tty flags word; see **Summary** below.

The terminal drivers provide for mapping between upper and lower case on terminals lacking lower case, and for other special processing on deficient terminals.

Finally, in the new terminal driver, there is an output flush character, normally ^O, which sets the LFLUSHO bit in the local mode word, causing subsequent output to be flushed until it is cleared by a program or more input is typed. This character has effect in both cooked and CBREAK modes and causes pending input to be retyped if there is any pending input. An ioctl to flush the characters in the input and output queues, TIOCFLUSH, is also available.

Upper case terminals and Hazeltines

If the LCASE bit is set in the tty flags, then all upper-case letters are mapped into the corresponding lower-case letter. The upper-case letter may be generated by preceding it by '\'. If the new terminal driver is being used, then upper case letters are preceded by a '\' when output. In addition, the following escape sequences can be generated on output and accepted on input:

for	`	|	~	{	}
use	\\'	\\!	\\^	\\(	\\)

To deal with Hazeltine terminals, which do not understand that ~ has been made into an ASCII character, the LTILDE bit may be set in the local mode word when using the new terminal driver; in this case the character ~ will be replaced with the character ` on output.

Flow control.

There are two characters (the stop character, normally ^S, and the start character, normally ^Q) which cause output to be suspended and resumed respectively. Extra stop characters typed when output is already stopped have no effect, unless the start and stop characters are made the same, in which case output resumes.

A bit in the flags word may be set to put the terminal into TANDEM mode. In this mode the system produces a stop character (default ^S) when the input queue is in danger of overflowing, and a start character (default ^Q) when the input has drained sufficiently. This mode is useful when the terminal is actually another machine that obeys the conventions.

Line control and breaks.

There are several *ioctl* calls available to control the state of the terminal line. The TIOCSBRK ioctl will set the break bit in the hardware interface causing a break condition to exist; this can be cleared (usually after a delay with *sleep*(3)) by TIOCCBRK. Break conditions in the input are reflected as a null character in RAW mode or as the interrupt character in cooked or CBREAK mode. The TIOCCDTR ioctl will clear the data terminal ready condition; it can be set again by TIOCSDTR.

When the carrier signal from the dataset drops (usually because the user has hung up his terminal) a SIGHUP hangup signal is sent to the processes in the distinguished process group of the terminal; this usually causes them to terminate (the SIGHUP can be suppressed by setting the LNOHANG bit in the local state word of the driver.) Access to the terminal by other processes is then normally revoked, so any further reads will fail, and programs that read a terminal and test for end-of-file on their input will terminate appropriately.

When using an ACU it is possible to ask that the phone line be hung up on the last close with the TIOCHPCL ioctl; this is normally done on the outgoing line.

Interrupt characters.

There are several characters that generate interrupts in cooked and CBREAK mode; all are sent to the processes in the control group of the terminal, as if a TIOCGPGRP ioctl were done to get the process group and then a *killpg*(2) system call were done, except that these characters also flush pending input and output when typed at a terminal (*à`'la* TIOCFLUSH). The characters shown here are the defaults; the field names in the structures (given below) are also shown. The characters may be changed.

^C **t_intrc** (ETX) generates a SIGINT signal. This is the normal way to stop a process which is no longer interesting, or to regain control in an interactive program.

^\ **t_quitc** (FS) generates a SIGQUIT signal. This is used to cause a program to terminate and produce a core image, if possible, in the file **core** in the current directory.

^Z **t_suspc** (EM) generates a SIGTSTP signal, which is used to suspend the current process group.

^Y **t_dsuspc** (SUB) generates a SIGTSTP signal as ^Z does, but the signal is sent when a program attempts to read the ^Y, rather than when it is typed.

Job access control.

When using the new terminal driver, if a process which is not in the distinguished process group of its control terminal attempts to read from that terminal its process group is sent a SIGTTIN signal. This signal normally causes the members of that process group to stop. If, however, the process is ignoring SIGTTIN, has SIGTTIN blocked, is an *orphan process*, or is in the middle of process creation using *vfork*(2)), it is instead returned an end-of-file. (An *orphan process* is a process whose parent has exited and has been inherited by the *init*(8) process.) Under older UNIX systems these processes would typically have had their input files reset to **/dev/null**, so this is a compatible change.

When using the new terminal driver with the LTOSTOP bit set in the local modes, a process is prohibited from writing on its control terminal if it is not in the distinguished process group for that terminal. Processes which are holding or ignoring SIGTTOU signals, which are orphans, or which are in the middle of a *vfork*(2) are excepted and allowed to produce output.

Summary of modes.

Unfortunately, due to the evolution of the terminal driver, there are 4 different structures which contain various portions of the driver data. The first of these (**sgttyb**) contains that part of the information largely common between version 6 and version 7 UNIX systems. The second contains additional control characters added in version 7. The third is a word of local state peculiar to the new terminal driver, and the fourth is another structure of special characters added for the new driver. In the future a single structure may be made available to programs which need to access all this information; most programs need not concern themselves with all this state.

Basic modes: sgtty.

The basic *ioctls* use the structure defined in *<sgtty.h>*:

```
struct sgttyb {
        char    sg_ispeed;
        char    sg_ospeed;
        char    sg_erase;
        char    sg_kill;
        short   sg_flags;
};
```

The *sg_ispeed* and *sg_ospeed* fields describe the input and output speeds of the device according to the following table, which corresponds to the DEC DH-11 interface. If other hardware is used, impossible speed changes are ignored. Symbolic values in the table are as defined in <*sgtty.h*>.

B0	0	(hang up dataphone)
B50	1	50 baud
B75	2	75 baud
B110	3	110 baud
B134	4	134.5 baud
B150	5	150 baud
B200	6	200 baud
B300	7	300 baud
B600	8	600 baud
B1200	9	1200 baud
B1800	10	1800 baud
B2400	11	2400 baud
B4800	12	4800 baud
B9600	13	9600 baud
EXTA	14	External A
EXTB	15	External B

In the current configuration, only 110, 150, 300 and 1200 baud are really supported on dial-up lines. Code conversion and line control required for IBM 2741's (134.5 baud) must be implemented by the user's program. The half-duplex line discipline required for the 202 dataset (1200 baud) is not supplied; full-duplex 212 datasets work fine.

The *sg_erase* and *sg_kill* fields of the argument structure specify the erase and kill characters respectively. (Defaults are DELETE and ^U.)

The *sg_flags* field of the argument structure contains several bits that determine the system's treatment of the terminal:

ALLDELAY	0177400	Delay algorithm selection
BSDELAY	0100000	Select backspace delays (not implemented):
BS0	0	
BS1	0100000	
VTDELAY	0040000	Select form-feed and vertical-tab delays:
FF0	0	
FF1	0100000	
CRDELAY	0030000	Select carriage-return delays:
CR0	0	
CR1	0010000	
CR2	0020000	
CR3	0030000	
TBDELAY	0006000	Select tab delays:
TAB0	0	
TAB1	0001000	
TAB2	0004000	
XTABS	0006000	

NLDELAY	0001400	Select new-line delays:
NL0	0	
NL1	0000400	
NL2	0001000	
NL3	0001400	
EVENP	0000200	Even parity allowed on input (most terminals)
ODDP	0000100	Odd parity allowed on input
RAW	0000040	Raw mode: wake up on all characters, 8-bit interface
CRMOD	0000020	Map CR into LF; echo LF or CR as CR-LF
ECHO	0000010	Echo (full duplex)
LCASE	0000004	Map upper case to lower on input
CBREAK	0000002	Return each character as soon as typed
TANDEM	0000001	Automatic flow control

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay.

Backspace delays are currently ignored but might be used for Terminet 300's.

If a form-feed/vertical tab delay is specified, it lasts for about 2 seconds.

Carriage-return delay type 1 lasts about .08 seconds and is suitable for the Terminet 300. Delay type 2 lasts about .16 seconds and is suitable for the VT05 and the TI 700. Delay type 3 is suitable for the concept-100 and pads lines to be at least 9 characters at 9600 baud.

New-line delay type 1 is dependent on the current column and is tuned for Teletype model 37's. Type 2 is useful for the VT05 and is about .10 seconds. Type 3 is unimplemented and is 0.

Tab delay type 1 is dependent on the amount of movement and is tuned to the Teletype model 37. Type 3, called XTABS, is not a delay at all but causes tabs to be replaced by the appropriate number of spaces on output.

Input characters with the wrong parity, as determined by bits 200 and 100, are ignored in cooked and CBREAK mode.

RAW disables all processing save output flushing with LFLUSHO; full 8 bits of input are given as soon as it is available; all 8 bits are passed on output. A break condition in the input is reported as a null character. If the input queue overflows in raw mode it is discarded; this applies to both new and old drivers.

CRMOD causes input carriage returns to be turned into new-lines; input of either CR or LF causes LF-CR both to be echoed (for terminals with a new-line function).

CBREAK is a sort of half-cooked (rare?) mode. Programs can read each character as soon as typed, instead of waiting for a full line; all processing is done except the input editing: character and word erase and line kill, input reprint, and the special treatment of \ or EOT are disabled.

TANDEM mode causes the system to produce a stop character (default ^S) whenever the input queue is in danger of overflowing, and a start character (default ^Q) when the input queue has drained sufficiently. It is useful for flow control when the 'terminal' is really another computer which understands the conventions.

Basic ioctls

In addition to the TIOCSETD and TIOCGETD disciplines discussed in **Line disciplines** above, a large number of other *ioctl*(2) calls apply to terminals, and have the general form:

#include <sgtty.h>

ioctl(fildes, code, arg)
struct sgttyb *arg;

The applicable codes are:

TIOCGETP Fetch the basic parameters associated with the terminal, and store in the pointed-to *sgttyb* structure.

TIOCSETP Set the parameters according to the pointed-to *sgttyb* structure. The interface delays until output is quiescent, then throws away any unread characters, before changing the modes.

TIOCSETN Set the parameters like TIOCSETP but do not delay or flush input. Input is not preserved, however, when changing to or from RAW.

With the following codes the *arg* is ignored.

TIOCEXCL Set "exclusive-use" mode: no further opens are permitted until the file has been closed.

TIOCNXCL Turn off "exclusive-use" mode.

TIOCHPCL When the file is closed for the last time, hang up the terminal. This is useful when the line is associated with an ACU used to place outgoing calls.

TIOCFLUSH All characters waiting in input or output queues are flushed.

The remaining calls are not available in vanilla version 7 UNIX. In cases where arguments are required, they are described; *arg* should otherwise be given as 0.

TIOCSTI the argument is the address of a character which the system pretends was typed on the terminal.

TIOCSBRK the break bit is set in the terminal.

TIOCCBRK the break bit is cleared.

TIOCSDTR data terminal ready is set.

TIOCCDTR data terminal ready is cleared.

TIOCGPGRP arg is the address of a word into which is placed the process group number of the control terminal.

TIOCSPGRP arg is a word (typically a process id) which becomes the process group for the control terminal.

FIONREAD returns in the long integer whose address is arg the number of immediately readable characters from the argument unit. This works for files, pipes, and terminals.

Tchars

The second structure associated with each terminal specifies characters that are special in both the old and new terminal interfaces: The following structure is defined in <*sys/ioctl.h*>, which is automatically included in <*sgtty.h*>:

```
struct tchars {
        char    t_intrc;        /* interrupt */
        char    t_quitc;        /* quit */
        char    t_startc;       /* start output */
        char    t_stopc;        /* stop output */
        char    t_eofc;         /* end-of-file */
        char    t_brkc;         /* input delimiter (like nl) */
};
```

The default values for these characters are ^C, ^\, ^Q, ^S, ^D, and -1. A character value of -1 eliminates the effect of that character. The *t_brkc* character, by default -1, acts like a new-line in that it terminates a 'line,' is echoed, and is passed to the program. The 'stop' and 'start' characters may be the same, to produce a toggle effect. It is probably counterproductive to make other special characters (including erase and kill) identical. The applicable ioctl calls are:

TIOCGETC Get the special characters and put them in the specified structure.

TIOCSETC Set the special characters to those given in the structure.

Local mode

The third structure associated with each terminal is a local mode word; except for the LNOHANG bit, this word is interpreted only when the new driver is in use. The bits of the local mode word are:

LCRTBS	000001	Backspace on erase rather than echoing erase
LPRTERA	000002	Printing terminal erase mode
LCRTERA	000004	Erase character echoes as backspace-space-backspace
LTILDE	000010	Convert ˜ to ` on output (for Hazeltine terminals)
LMDMBUF	000020	Stop/start output when carrier drops
LLITOUT	000040	Suppress output translations
LTOSTOP	000100	Send SIGTTOU for background output
LFLUSHO	000200	Output is being flushed
LNOHANG	000400	Don't send hangup when carrier drops
LETXACK	001000	Diablo style buffer hacking (unimplemented)
LCRTKIL	002000	BS-space-BS erase entire line on line kill
LCTLECH	010000	Echo input control chars as ^X, delete as ^?
LPENDIN	020000	Retype pending input at next read or input character
LDECCTQ	040000	Only ^Q restarts output after ^S, like DEC systems

The applicable *ioctl* functions are:

TIOCLBIS arg is the address of a mask which is the bits to be set in the local mode word.

TIOCLBIC arg is the address of a mask of bits to be cleared in the local mode word.

TIOCLSET arg is the address of a mask to be placed in the local mode word.

TIOCLGET arg is the address of a word into which the current mask is placed.

Local special chars

The final structure associated with each terminal is the *ltchars* structure which defines interrupt characters for the new terminal driver. Its structure is:

```
struct ltchars {
        char    t_suspc;        /* stop process signal */
        char    t_dsuspc;       /* delayed stop process signal */
        char    t_rprntc;       /* reprint line */
        char    t_flushc;       /* flush output (toggles) */
        char    t_werasc;       /* word erase */
        char    t_lnextc;       /* literal next character */
};
```

The default values for these characters are ^Z, ^Y, ^R, ^O, ^W, and ^V. A value of –1 disables the character.

The applicable *ioctl* functions are:

TIOCSLTC args is the address of a *ltchars* structure which defines the new local special characters.

TIOCGLTC args is the address of a *ltchars* structure into which is placed the current set of local special characters.

FILES

/dev/tty
/dev/tty*
/dev/console

SEE ALSO

csh(1), stty(1), ioctl(2), sigvec(2), stty(3C), getty(8), init(8)

BUGS

Half-duplex terminals are not supported.

NAME

udp – Internet User Datagram Protocol

SYNOPSIS

None; comes automatically with *inet*(4F).

DESCRIPTION

The User Datagram Protocol (UDP) is defined to make available a datagram mode of packet switched computer communicaton in the environment of an interconnected set of computer networks. The protocol assumes that the Internet Protocol (IP) as described in *ip*(4P) is used as the underlying protocol.

The protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) as described in *tcp*(4P).

The UNIX implementation of UDP makes it available as a socket of type SOCK_DGRAM. UDP sockets are normally used in a connectionless fashion, with the *sendto* and *recvfrom* calls described in *send*(2) and *recv*(2).

A UDP socket is created with a *socket*(2) call:

s = socket(AF_INET, SOCK_DGRAM, 0);

The socket initially has no address associated with it, and may be given an address with a *bind*(2) call as described in *inet*(4F). If no *bind* call is done, then the address assignment procedure described in *inet*(4F) is repeated as each datagram is sent.

When datagrams are sent the system encapsulates the user supplied data with UDP and IP headers. Unless the invoker is the super-user datagrams which would become broadcast packets on the network to which they are addressed are not allowed. Unless the socket has had a SO_DONTROUTE option enabled (see *socket*(2)) the outgoing datagram is routed through the routing tables as described in *routing*(4N). If there is insufficient system buffer space to temporarily hold the datagram while it is being trasmitted, the *sendto* may result in a ENOBUFS error. Other errors (ENETUNREACH, EADDRNOTAVAIL, EACCES, EMSGSIZE) may be generated by *icmp*(4P) or by the network interfaces themselves, and are reflected back in the *send* call.

As each UDP datagram arrives at a host the system strips out the IP options and checksums the data field, discarding the datagram if the checksum indicates that the datagram has been damaged. If no socket exists for the datagram to be sent to then an ICMP error is returned to the originating socket. If a socket exists for this datagram to be sent to, then we will append the datagram and the address from which it came to a queue associated with the datagram socket. This queue has limited capacity (2048 bytes of datagrams) and arriving datagrams which will not fit within its *high-water* capacity are silently discarded.

UDP processes ICMP errors reflected to it by *icmp*(4P). QUENCH errors are ignored (this is well considered a bug); UNREACH, TIMXCEED and PARAMPROB errors cause the socket to be disconnected from its peer if it was bound to a peer using *bind*(2) so that subsequent attempts to send datagrams via that socket will give an error indication.

The UDP datagram protocol differs from IP datagrams in that it adds a checksum over the data bytes and contains a 16-bit socket address on each machine rather than just the 32-bit machine address; UDP datagrams are addressed to sockets; IP packets are addressed to hosts.

SEE ALSO

recv(2), send(2), inet(4F)
"User Datagram Protocol", RFC768, John Postel, USC-ISI (Sun 800-1054-01)

BUGS

SIOCSHIWAT and SIOCGHIWAT ioctl's to set and get the high water mark for the socket queue, and so that it can be changed from 2048 bytes to be larger or smaller, have been defined (in <sys/ioctl.h>) but not implemented.

Something sensible should be done with QUENCH errors if the socket is bound to a peer socket.

NAME

vp – Ikon 10071-5 Multibus Versatec parallel printer interface

SYNOPSIS

device vp0 at mb0 csr 0x400 priority 2

DESCRIPTION

The Sun Multibus interface to the Versatec printer/plotter is supported by the Ikon parallel interface board, a word DMA device, which is output only.

The Versatec is normally handled by the line printer spooling system and should not be accessed by the user directly.

Opening the device */dev/vp0* may yield one of two errors: ENXIO indicates that the device is already in use. EIO indicates that the device is offline.

The printer operates in either print or plot mode. To set the printer into plot mode you should include <vcmd.h> and use the *ioctl*(2) call

```
ioctl(f, VSETSTATE, plotmd);
```

where *plotmd* is defined to be

```
int plotmd[] = { VPLOT, 0, 0 };
```

When going back into print mode from plot mode you normally eject paper by sending it an EOT after putting into print mode:

```
int prtmd[] = { VPRINT, 0, 0 };
...
fflush(vp);
ioctl(f, VSETSTATE, prtmd);
write(f, "\04", 1);
```

FILES

/dev/vp0

SEE ALSO

Multibus/Versatec Interface, Ikon Corp (Includes Versatec Manual) (Sun 800-1065-01)

BUGS

If you use the standard i/o library on the Versatec, be sure to explicitly set a buffer using *setbuf*, since the library will not use buffered output by default, and will run very slowly.

This driver is not supported.

Writes must start on even byte boundaries and be an even number of bytes in length.

NAME

vpc – Systech VPC-2200 Versatec printer/plotter and Centronics printer interface

SYNOPSIS

device vpc0 at mb0 csr 0x480 priority 2

DESCRIPTION

The Sun Multibus interface to the Versatec printer/plotter and to Centronics printers is supported by the Systech parallel interface board, an output-only byte-wide DMA device. The device has one channel for Versatec devices and one channel for Centronics devices, with an optional long lines interface for Versatec devices.

Devices attached to this interface are normally handled by the line printer spooling system and should not be accessed by the user directly.

Opening the devices */dev/vp0 /dev/lp0* may yield one of two errors: ENXIO indicates that the device is already in use. EIO indicates that the device is offline.

The Versatec printer/plotter operates in either print or plot mode. To set the printer into plot mode you should include <vcmd.h> and use the *ioctl*(2) call:

```
ioctl(f, VSETSTATE, plotmd);
```

where *plotmd* is defined to be

```
int plotmd[] = { VPLOT, 0, 0 };
```

When going back into print mode from plot mode you normally eject paper by sending it an EOT after putting into print mode:

```
int prtmd[] = { VPRINT, 0, 0 };
...
fflush(vpc);
ioctl(f, VSETSTATE, prtmd);
write(f, "\04", 1);
```

FILES

/dev/vp0
/dev/lp0

SEE ALSO

Systech VPC-2200 Versatec Printer/Plotter Controller Technical Manual

BUGS

If you use the standard I/O library on the Versatec, be sure to explicitly set a buffer using *setbuf*, since the library will not use buffered output by default, and will run very slowly.

Currently only 8 bit I/O is supported in the driver, even though the device supports 16 bit I/O.

NAME

win – Sun window system

SYNOPSIS

pseudo-device win128
pseudo-device dtop4

DESCRIPTION

The *win* device accesses the system drivers supporting the Sun window system.

Each window in the system is represented by a /dev/win* device. The windows are organized as a tree with windows being subwindows of their parents, and covering/covered by their siblings. Each window has a position in the tree, a position on a display screen, an input queue, and information telling what parts of it are exposed.

The window driver multiplexes keyboard and mouse input among the several windows, tracks the mouse with a cursor on the screen, provides each window access to information about what parts of it are exposed, and notifies the manager process for a window when the exposed area of the window changes so that the window may repair its display.

The **dtop4** pseudo device line in a kernel configuration file indicates the number of separate "desktops" (frame buffers) that can be actively running the Sun window system at once.

Full information on the window system functions is given in the *Programmer's Reference Manual for SunWindows.*

FILES

/dev/win[0-9]
/dev/win[0-9][0-9]

SEE ALSO

Programmer's Reference Manual for SunWindows

NAME

xy – Disk driver for Xylogics SMD Disk Controllers

SYNOPSIS

controller xyc0 at mb0 csr 0xee40 priority 2
disk xy0 at xyc0 drive 0
disk xy1 at xyc0 drive 1

DESCRIPTION

Files with minor device numbers 0 through 7 refer to various portions of drive 0; minor devices 8 through 15 refer to drive 1, and so on. The standard device names begin with "xy" followed by the drive number and then a letter a-h for partitions 0-7 respectively. The character ? stands here for a drive number in the range 0-7.

The block file's access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.'

In raw I/O counts should be a multiple of 512 bytes (a disk sector). Likewise *seek* calls should specify a multiple of 512 bytes.

DISK SUPPORT

This driver handles all SMD drives, by reading a label from sector 0 of the drive which describes the disk geometry and partitioning.

The xy?a partition is normally used for the root file system on a disk, the xy?b partition as a paging area, and the xy?c partition for pack-pack copying (it normally maps the entire disk). The rest of the disk is normally the xy?h partition.

FILES

/dev/xy[0-7][a-h]	block files
/dev/rxy[0-7][a-h]	raw files

SEE ALSO

dkio(4S), xy(4S)
Xylogics Model 440 Peripheral Processor SMD Disk Subsystem Maintenance and Reference Manual (Sun 800-1005-01)
Xylogics Model 450 Peripheral Processor SMD Disk Subsystem Maintenance and Reference Manual (Sun 800-1025-01)

DIAGNOSTICS

xyc%d: self test error %x - %s. Self test error in controller, see the Maintenance and Reference Manual.

xyc%d: address mode jumper is wrong. The controller is strapped for 24-bit Multibus addresses; the Sun uses 20-bit addresses. See the Hardware Configuration and Expansion section of the System Manager's Manual for your Sun Workstaiton for instructions on setting the jumpers on the 450.

xyattach: can't get bad sector info. The bad sector forwarding information for the disk, which is kept on the last cylinder, could not be read.

xy%d: drive type %d clash with xy%d. The 450 does not support mixing the drive types found on these units on a single controller.

xy%d: initialization failed.

xy%d: error %x reading label on head %d. Error reading drive geometry/partition table information.

xy%d: Corrupt label. The geometry/partition label checksum was incorrect.

xy%d: Unsupported phys partition # %d.

xy%d: offline.

xy%d%c: *cmd how (msg)* **blk %d.** A command such as read, write, or format encountered a error condition (how): either it *failed,* the unit was *restored,* or an operation was *retry*'ed. The *msg* is derived from the error number given by the controller, indicating a condition such as "drive not ready", "sector not found" or "disk write protected".

BUGS

In raw I/O *read* and *write*(2) truncate file offsets to 512-byte block boundaries, and *write* scribbles on the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, *read, write* and *lseek*(2) should always deal in 512-byte multiples.

NAME

zs – zilog 8530 SCC serial comunications driver

SYNOPSIS

device zs0 at mb0 csr 0xf5a000 flags 3 priority 6

DESCRIPTION

The Zilog 8530 provides 2 serial communication lines with full modem control. Each line behaves as described in *tty*(4). Input and output for each line may independently be set to run at any of 16 speeds; see *tty*(4) for the encoding.

FILES

/dev/tty[a-d]

SEE ALSO

tty(4)
Zilog Z8030/Z8530 SCC Serial Communications Controller (Sun 800-1052-01)

DIAGNOSTICS

zs%d%c: silo overflow. The character input silo overflowed before it could be serviced.

NAME

a.out – assembler and link editor output

SYNOPSIS

#include <a.out.h>
#include <stab.h>
#include <nlist.h>

DESCRIPTION

A.out is the output file of the assembler *as*(1S) and the link editor *ld*(1). The latter makes *a.out* executable if there were no errors and no unresolved external references. Layout information as given in the include file for the Sun system is:

```
/*
 * Header prepended to each a.out file.
 */
struct exec {
        long      a_magic;  /* magic number */
        unsigned  a_text;   /* size of text segment */
        unsigned  a_data;   /* size of initialized data */
        unsigned  a_bss;    /* size of uninitialized data */
        unsigned  a_syms;   /* size of symbol table */
        unsigned  a_entry;  /* entry point */
        unsigned  a_trsize; /* size of text relocation */
        unsigned  a_drsize; /* size of data relocation */
};

#define  OMAGIC 0407        /* old impure format */
#define  NMAGIC 0410        /* read-only text */
#define  ZMAGIC 0413        /* demand load format */

#define  PAGSIZ  2048
#define  SEGSIZ  0x8000
#define  TXTRELOC SEGSIZ
/*
 * Macros which take exec structures as arguments and tell whether
 * the file has a reasonable magic number or offsets to text|symbols|strings.
 */
#define  N_BADMAG(x) \
   (((x).a_magic)!=OMAGIC && ((x).a_magic)!=NMAGIC && ((x).a_magic)!=ZMAGIC)

#define  N_TXTOFF(x) \
        ((x).a_magic==ZMAGIC ? PAGSIZ : sizeof (struct exec))
#define N_SYMOFF(x) \
        (N_TXTOFF(x) +  (x).a_text+ (x).a_data +  (x).a_trsize+ (x).a_drsize)
#define  N_STROFF(x) \
        (N_SYMOFF(x) +  (x).a_syms)
/*
 * Macros which take exec structures as arguments and tell where the
 * various pieces will be loaded.
 */
#define N_TXTADDR(x) TXTRELOC
#define N_DATADDR(x) \
        (((x).a_magic==OMAGIC)? (N_TXTADDR(x)+ (x).a_text) \
        : (SEGSIZ+ ((N_TXTADDR(x)+ (x).a_text-1) & ~SEGRND)))
```

```
#define N_BSSADDR(x)  (N_DATADDR(x)+ (x).a_data)
```

The *a.out* file has five sections: a header, the program text and data, relocation information, a symbol table and a string table (in that order). The last three may be omitted if the program was loaded with the '–s' option of *ld* or if the symbols and relocation have been removed by *strip*(1).

In the header the sizes of each section are given in bytes. The size of the header is not included in any of the other sizes.

When an *a.out* file is executed, three logical segments are set up: the text segment, the data segment (with uninitialized data, which starts off as all 0, following initialized data), and a stack. The header is not loaded with the text segment. If the magic number in the header is OMAGIC (0407), it means that this is a non-sharable text which is not to be write-protected, so the data segment is immediately contiguous with the text segment. This is rarely used. If the magic number is NMAGIC (0410) or ZMAGIC (0413), the data segment begins at the first segment boundary following the text segment, and the text segment is not writable by the program; other processes executing the same file will share the text segment. For ZMAGIC format, the text segment begins on a page boundary in the *a.out* file; the remaining bytes after the header in the first block are reserved and should be zero. In this case the text and data sizes must both be multiples of the page size, and the pages of the file will be brought into the running image as needed, and not pre-loaded as with the other formats. This is especially suitable for very large programs and is the default format produced by *ld*(1). The macros N_TXTADDR, N_DATADDR, and N_BSSADDR give the core addresses at which the text, data, and bss segments, respectively, will be loaded.

The stack starts at the highest possible location in the memory image, and grows downwards. The stack is automatically extended as required. The data segment is extended as requested by *brk*(2) or *sbrk*(2).

After the header in the file follow the text, data, text relocation data relocation, symbol table and string table in that order. The text begins at byte PAGSIZ in the file for ZMAGIC format or just after the header for the other formats. The N_TXTOFF macro returns this absolute file position when given the name of an exec structure as argument. The data segment is contiguous with the text and immediately followed by the text relocation and then the data relocation information. The symbol table follows all this; its position is computed by the N_SYMOFF macro. Finally, the string table immediately follows the symbol table at a position which can be gotten easily using N_STROFF. The first 4 bytes of the string table are not used for string storage, but rather contain the size of the string table; this size INCLUDES the 4 bytes, the minimum string table size is thus 4.

RELOCATION

The value of a byte in the text or data which is not a portion of a reference to an undefined external symbol is exactly that value which will appear in memory when the file is executed. If a byte in the text or data involves a reference to an undefined external symbol, as indicated by the relocation information, then the value stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol is added to the bytes in the file.

If relocation information is present, it amounts to eight bytes per relocatable datum as in the following structure:

```
/*
 * Format of a relocation datum.
 */
struct relocation_info {
        int          r_address;          /* address which is relocated */
```

```
        unsigned r_symbolnum:24, /* local symbol ordinal */
                 r_pcrel:1,      /* was relocated pc relative already */
                 r_length:2,     /* 0=byte, 1=word, 2=long */
                 r_extern:1,     /* does not include value of sym referenced */
                 :4;             /* nothing, yet */
};
```

There is no relocation information if a_trsize+ a_drsize==0. If r_extern is 0, then r_symbolnum is actually a n_type for the relocation (i.e. N_TEXT meaning relative to segment text origin.)

SYMBOL TABLE

The layout of a symbol table entry and the principal flag values that distinguish symbol types are given in the include file as follows:

```
/*
 * Format of a symbol table entry.
 */
struct nlist {
        union {
            char     *n_name; /* for use when in-memory */
            long     n_strx;  /* index into file string table */
        } n_un;
        unsigned char n_type; /* type flag, i.e. N_TEXT etc; see below */
        char         n_other;
        short        n_desc;  /* see <stab.h> */
        unsigned     n_value; /* value of this symbol (or adb offset) */
};
#define n_hash       n_desc   /* used internally by ld */

/*
 * Simple values for n_type.
 */
#define N_UNDF      0x0       /* undefined */
#define N_ABS       0x2       /* absolute */
#define N_TEXT      0x4       /* text */
#define N_DATA      0x6       /* data */
#define N_BSS       0x8       /* bss */
#define N_COMM      0x12      /* common (internal to ld) */
#define N_FN        0x1f      /* file name symbol */

#define N_EXT       01        /* external bit, or'ed in */
#define N_TYPE      0x1e      /* mask for all the type bits */

/*
 * Other permanent symbol table entries have some of the N_STAB bits set.
 * These are given in <stab.h>
 */
#define N_STAB      0xe0      /* if any of these bits set, don't discard */
```

In the *a.out* file a symbol's n_un.n_strx field gives an index into the string table. A n_strx value of 0 indicates that no name is associated with a particular symbol table entry. The field n_un.n_name can be used to refer to the symbol name only if the program sets this up using n_strx and appropriate data from the string table. Because of the union in the nlist declaration, it is impossible in C to statically initialize such a structure. If this must be done (as when using *nlist*(3)) the file **<nlist.h>** should be included, rather that **<a.out.h>**; this contains the declaration without the union.

If a symbol's type is undefined external, and the value field is non-zero, the symbol is interpreted by the loader *ld* as the name of a common region whose size is indicated by the value of the symbol.

STAB SYMBOLS

Stab.h defines some values of the n_type field of the symbol table of a.out files. These are the types for permanent symbols (that is, not local labels, etc.) used by the debuggers *adb*(1S) and *dbx*(1) and the Berkeley Pascal compiler *pc*(1). Symbol table entries can be produced by the *.stabs* assembler directive. This allows one to specify a double-quote delimited name, a symbol type, one char and one short of information about the symbol, and an unsigned long (usually an address). To avoid having to produce an explicit label for the address field, the *.stabd* directive can be used to implicitly address the current location. If no name is needed, symbol table entries can be generated using the *.stabn* directive. The loader promises to preserve the order of symbol table entries produced by *.stab* directives.

The n_value field of a symbol is relocated by the link editor as an address within the appropriate segment. N_value fields of symbols not in any segment are unchanged by the linker. In addition, the linker will discard certain symbols, according to rules of its own, unless the n_type field has one of the bits masked by N_STAB set.

This allows up to 112 (7 * 16) symbol types, split between the various segments. Some of these have already been claimed. The debugger, *adb*(1S), uses the following n_type values:

```
#define N_GSYM   0x20 /* global symbol: name,,0,type,0 */
#define N_FNAME  0x22 /* procedure name (f77 kludge): name,,0 */
#define N_FUN    0x24 /* procedure: name,,0,linenumber,address */
#define N_STSYM  0x26 /* static symbol: name,,0,type,address */
#define N_LCSYM  0x28 /* .lcomm symbol: name,,0,type,address */
#define N_RSYM   0x40 /* register sym: name,,0,type,register */
#define N_SLINE  0x44 /* src line: 0,,0,linenumber,address */
#define N_SSYM   0x60 /* structure elt: name,,0,type,struct_offset */
#define N_SO     0x64 /* source file name: name,,0,0,address */
#define N_LSYM   0x80 /* local sym: name,,0,type,offset */
#define N_SOL    0x84 /* #included file name: name,,0,0,address */
#define N_PSYM   0xa0 /* parameter: name,,0,type,offset */
#define N_ENTRY  0xa4 /* alternate entry: name,linenumber,address */
#define N_LBRAC  0xc0 /* left bracket: 0,,0,nesting level,address */
#define N_RBRAC  0xe0 /* right bracket: 0,,0,nesting level,address */
#define N_BCOMM  0xe2 /* begin common: name,, */
#define N_ECOMM  0xe4 /* end common: name,, */
#define N_ECOML  0xe8 /* end common (local name): ,,address */
#define N_LENG   0xfe /* second stab entry with length information */
```

where the comments give the *adb* conventional use for *.stabs* and the n_name, n_other, n_desc, and n_value fields of the given n_type. *Adb* uses the n_desc field to hold a type specifier in the form used by the Portable C Compiler, *cc*(1), in which a base type is qualified in the following structure:

```
struct desc {
        short  q6:2,
               q5:2,
               q4:2,
               q3:2,
               q2:2,
               q1:2,
```

```
		basic:4;
};
```

There are four qualifications, with q1 the most significant and q6 the least significant:

0	none
1	pointer
2	function
3	array

The sixteen basic types are assigned as follows:

0	undefined
1	function argument
2	character
3	short
4	int
5	long
6	float
7	double
8	structure
9	union
10	enumeration
11	member of enumeration
12	unsigned character
13	unsigned short
14	unsigned int
15	unsigned long

The Berkeley Pascal compiler, *pc*(1), uses the following n_type value:

```
#define N_PC   0x30   /* global pascal symbol: name,,0,subtype,line */
```

and uses the following subtypes to do type checking across separately compiled files:

1	source file name
2	included file name
3	global label
4	global constant
5	global type
6	global variable
7	global function
8	global procedure
9	external function
10	external procedure
11	library variable
12	library routine

The new *dbx*(1) debugger uses an entirely different interpretation for the *stabs* symbol-table entries. Currently, this is understood only by *dbx* and *cc*, but its use should supplant the current interpretation as soon as *adb* and *pc* can be modified to use it.

SEE ALSO

adb(1S), as(1S), ld(1), nm(1), dbx(1), strip(1)

BUGS

There are currently two interpretations of the *stabs* symbol–table information. This creates great confusion when trying to build a program for debugging.

Due to the amount of symbolic information necessary for high–level debugging, the whole *a.out* structure has been streched well beyond its original design, and should be replaced by something with a more sophisticated symbol–table mechanism. The demands of future languages will only compound the problems.

NAME

aliases – aliases file for sendmail

SYNOPSIS

/usr/lib/aliases
/usr/lib/aliases.dir
/usr/lib/aliases.pag

DESCRIPTION

These files describe user id aliases used by */usr/lib/sendmail*. */usr/lib/aliases* is formatted as a series of lines of the form

name: name_1, name2, name_3, . . .

The *name* is the name to alias, and the *name_n* are the aliases for that name. Lines beginning with white space are continuation lines. Lines beginning with '#' are comments.

Aliasing occurs only on local names. Loops can not occur, since no message will be sent to any person more than once.

After aliasing has been done, local and valid recipients who have a ".forward" file in their home directory have messages forwarded to the list of users defined in that file.

/usr/lib/aliases is only the raw data file; the actual aliasing information is placed into a binary format in the files */usr/lib/aliases.dir* and */usr/lib/aliases.pag* using the program *newaliases*(8). A *newaliases* command should be executed each time that */usr/lib/aliases* is changed for the change to take effect.

Several kinds of *name*'s are special:

owner–mary: fred

any errors resulting from a mail to *mary* are directed to *fred* instead of back to the person who sent the message. This is most useful when *mary* is a mailing list rather than an individual.

beer: :include:/usr/cyndi/beer;

All colons and semicolons are required as shown. The list of names in */usr/cyndi/beer* is included in the *name_n* list for the *beer* alias, in addition to any other names in the *name_n* list. This mechanism is for setting up a mailing list so that */usr/lib/aliases* doesn't have to be changed when people are added to or removed from the list. The included file (that is, */usr/cyndi/beer* in this case) may be changed at any time, and changes take effect immediately.

SEE ALSO

newaliases(8), dbm(3X), sendmail(8)
SENDMAIL Installation and Operation Guide.
SENDMAIL An Internetwork Mail Router.

BUGS

Because of restrictions in *dbm*(3X) a single alias cannot contain more than about 1000 bytes of information. You can get longer aliases by "chaining"; that is, make the last name in the alias be a dummy name which is a continuation alias.

NAME

ar – archive (library) file format

SYNOPSIS

#include <ar.h>

DESCRIPTION

The archive command *ar* combines several files into one. Archives are used mainly as libraries to be searched by the link-editor *ld.*

A file produced by *ar* has a magic string at the start, followed by the constituent files, each preceded by a file header. The magic number and header layout as described in the include file are:

```
/*	@(#)ar.h 1.1 83/08/01 SMI; from UCB 4.1 83/05/03*/

#define ARMAG	"!<arch>\n"
#define SARMAG 8

#define ARFMAG "`\n"

struct ar_hdr {
	char	ar_name[16];
	char	ar_date[12];
	char	ar_uid[6];
	char	ar_gid[6];
	char	ar_mode[8];
	char	ar_size[10];
	char	ar_fmag[2];
};
```

The name is a blank-padded string. The *ar_fmag* field contains ARFMAG to help verify the presence of a header. The other fields are left-adjusted, blank-padded numbers. They are decimal except for *ar_mode,* which is octal. The date is the modification date of the file at the time of its insertion into the archive.

Each file begins on a even (0 mod 2) boundary; a new-line is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

There is no provision for empty areas in an archive file.

The encoding of the header is portable across machines. If an archive contains printable files, the archive itself is printable.

SEE ALSO

ar(1), ld(1), nm(1)

BUGS

File names lose trailing blanks. Most software dealing with archives takes even an included blank as a name terminator.

NAME

core – format of memory image file

SYNOPSIS

#include <machine/param.h>

DESCRIPTION

The UNIX System writes out a memory image of a terminated process when any of various errors occur. See *sigvec*(2) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The memory image is called 'core' and is written in the process's working directory (provided it can be; normal access controls apply).

The maximum size of a *core* file is limited by *setrlimit*(2). Files which would be larger than the limit are not created.

Set-user-id programs do not produce core files when they terminate as this would be a security loophole.

The core file consists of the *u.* area, whose size (in pages) is defined by the UPAGES manifest in the *<machine/param.h>* file. The *u.* area starts with a *user* structure as given in *<sys/user.h>*. The remainder of the core file consists first of the data pages and then the stack pages of the process image. The amount of data space image in the core file is given (in pages) by the variable *u_dsize* in the *u.* area. The amount of stack image in the core file is given (in pages) by the variable *u_ssize* in the *u.* area.

SEE ALSO

adb(1), dbx(1), sigvec(2), setrlimit(2)

NAME

cpio – format of cpio archive

DESCRIPTION

The old format *header* structure, when the **c** option is not used, is:

```
struct {
        short  h_magic,
               h_dev,
               h_ino,
               h_mode,
               h_uid,
               h_gid,
               h_nlink,
               h_rdev,
               h_mtime[2],
               h_namesize,
               h_filesize[2];
        char   h_name[h_namesize rounded to a word];
} Hdr;
```

but note that the byte order here is that of the PDP-11 and the VAX, and that for the Sun you have to use *swab*(3) after reading and before writing headers.

When the **c** option is used, the *header* information is described by the statement below:

```
sscanf(Chdr, "%6o%6o%6o%6o%6o%6o%6o%6o%11lo%6o%6o%s",
        &Hdr.h_magic, &Hdr.h_dev, &Hdr.h_ino, &Hdr.h_mode,
        &Hdr.h_uid, &Hdr.h_gid, &Hdr.h_nlink, &Hdr.h_rdev,
        &Hdr.h_mtime, &Hdr.h_namesize, &Hdr.h_filesize, &Hdr.h_name);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file is recorded in an element of the array of varying length structures, *archive*, together with other items describing the file. Every instance of *h_magic* contains the constant 070707 (octal). The items *h_dev* through *h_mtime* have meanings explained in *stat*(2). The length of the null-terminated path name *h_name*, including the null byte, is given by *h_namesize*.

The last record of the *archive* always contains the name TRAILER!!!. Special files, directories, and the trailer, are recorded with *h_filesize* equal to zero.

SEE ALSO

cpio(1), find(1), stat(2)

NAME

crontab – table of times to run periodic jobs

DESCRIPTION

The */etc/cron* utility is a permanent process, started by */etc/rc.local*, that wakes up once every minute. */etc/cron* consults the file */usr/lib/crontab* to find out what tasks are to be done, and at what time.

Each line in */usr/lib/crontab* consists of six fields, separated by spaces or tabs, as follows:

1. minutes field, which can have values in the range 0 through 59.
2. hours field, which can have values in the range 0 through 23.
3. day of the month, in the range 1 through 31.
4. month of the year, in the range 1 through 12.
5. day of the week, in the range 1 through 7. Monday is day 1 in this scheme of things.
6. (the remainder of the line) is the command to be run. A percent character in this field is translated to a new-line character. Only the first line (up to a % or end of line) of the command field is executed by the Shell. The other lines are made available to the command as standard input.

Any of fields 1 through 5 can be a list of values separated by commas. A field can be a pair of numbers separated by a hyphen, indicating that the job is to be done for all the times in the specified range. If a field is an asterisk character (*) it means that the job is done for all possible values of the field.

FILES

/usr/lib/crontab

SEE ALSO

cron(8), rc(8)

EXAMPLE

```
0 0 * * * calendar -
15 0 * * * /etc/sa -s >/dev/null
15 4 * * * find /usr/preserve -mtime +7 -a -exec rm -f {} ;
40 4 * * * find / -name '#*' -atime +3 -exec rm -f {} ;
0,15,30,45 * * * * /etc/atrun
0,10,20,30,40,50 * * * * /etc/dmesg - >>/usr/adm/messages
5 4 * * * sh /etc/newsyslog
```

The *calendar* command run at minute 0 of hour 0 (midnight) of every day. The */etc/sa* command runs at 15 minutes after midnight every day. The two *find* commands run at 15 minutes past four and at 40 minutes past four, respectively, every day of the year. The *atrun* command (which processes shell scripts users have set up with *at*) runs every 15 minutes. The */etc/dmesg* command appends kernel messages to the */usr/adm/messages* file every ten minutes, and finally, the */usr/adm/syslog* script runs at five minutes after four every day.

NAME

dir – format of directories

SYNOPSIS

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

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry; see *fs*(5). The structure of a directory entry as given in the include file is:

```
/*
 * A directory consists of some number of blocks of DIRBLKSIZ
 * bytes, where DIRBLKSIZ is chosen such that it can be transferred
 * to disk in a single atomic operation (e.g. 512 bytes on most machines).
 *
 * Each DIRBLKSIZ byte block contains some number of directory entry
 * structures, which are of variable length. Each directory entry has
 * a struct direct at the front of it, containing its inode number,
 * the length of the entry, and the length of the name contained in
 * the entry. These are followed by the name padded to a 4 byte boundary
 * with null bytes. All names are guaranteed null terminated.
 * The maximum length of a name in a directory is MAXNAMLEN.
 *
 * The macro DIRSIZ(dp) gives the amount of space required to represent
 * a directory entry. Free space in a directory is represented by
 * entries which have dp->d_reclen > DIRSIZ(dp). All DIRBLKSIZ bytes
 * in a directory block are claimed by the directory entries. This
 * usually results in the last entry in a directory having a large
 * dp->d_reclen. When entries are deleted from a directory, the
 * space is returned to the previous entry in the same directory
 * block by increasing its dp->d_reclen. If the first entry of
 * a directory block is free, then its dp->d_ino is set to 0.
 * Entries other than the first in a directory do not normally have
 * dp->d_ino set to 0.
 */
#ifdef KERNEL
#define DIRBLKSIZ DEV_BSIZE
#else
#define  DIRBLKSIZ 512
#endif

#define MAXNAMLEN 255

/*
 * The DIRSIZ macro gives the minimum record length which will hold
 * the directory entry. This requires the amount of space in struct direct
 * without the d_name field, plus enough space for the name with a terminating
 * null byte (dp->d_namlen+1), rounded up to a 4 byte boundary.
 */
#undef DIRSIZ
#define DIRSIZ(dp)     ((sizeof (struct direct) - (MAXNAMLEN+1)) + (((dp)->d_namlen+1 + 3)&

struct    direct {
```

```
        u_long      d_ino;
        short       d_reclen;
        short       d_namlen;
        char        d_name[MAXNAMLEN + 1];
        /* typically shorter */
};

struct _dirdesc {
        int         dd_fd;
        long        dd_loc;
        long        dd_size;
        char        dd_buf[DIRBLKSIZ];
};
```

By convention, the first two entries in each directory are for '.' and '..'. The first is an entry for the directory itself. The second is for the parent directory. The meaning of '..' is modified for the root directory of the master file system ("/"), where '..' has the same meaning as '.'.

SEE ALSO

fs(5), readdir(3)

NAME

dump, dumpdates – incremental dump format

SYNOPSIS

#include <sys/types.h>
#include <sys/inode.h>
#include <dumprestor.h>

DESCRIPTION

Tapes used by *dump* and *restore*(8) contain:

a header record
two groups of bit map records
a group of records describing directories
a group of records describing files

The format of the header record and of the first record of each description as given in the include file *<dumprestor.h>* is:

```
#define NTREC       10
#define MLEN        16
#define MSIZ        4096

#define TS_TAPE     1
#define TS_INODE    2
#define TS_BITS     3
#define TS_ADDR     4
#define TS_END      5
#define TS_CLRI     6
#define MAGIC       (int) 60011
#define CHECKSUM    (int) 84446

struct  spcl {
        int         c_type;
        time_t      c_date;
        time_t      c_ddate;
        int         c_volume;
        daddr_t     c_tapea;
        ino_t       c_inumber;
        int         c_magic;
        int         c_checksum;
        struct      dinode          c_dinode;
        int         c_count;
        char        c_addr[BSIZE];
} spcl;

struct  idates {
        char        id_name[16];
        char        id_incno;
        time_t      id_ddate;
};

#define DUMPOUTFMT      "%-16s %c %s"          /* for printf */
                                               /* name, incno, ctime(date) */
#define DUMPINFMT       "%16s %c %[^\n]\n"     /* inverse for scanf */
```

NTREC is the default number of 1024 byte records in a physical tape block, changeable by the b option to *dump*. MLEN is the number of bits in a bit map word. MSIZ is the number of bit map words.

The TS_ entries are used in the *c_type* field to indicate what sort of header this is. The types and their meanings are as follows:

TS_TAPE	Tape volume label
TS_INODE	A file or directory follows. The *c_dinode* field is a copy of the disk inode and contains bits telling what sort of file this is.
TS_BITS	A bit map follows. This bit map has a one bit for each inode that was dumped.
TS_ADDR	A subrecord of a file description. See *c_addr* below.
TS_END	End of tape record.
TS_CLRI	A bit map follows. This bit map contains a zero bit for all inodes that were empty on the file system when dumped.
MAGIC	All header records have this number in *c_magic*.
CHECKSUM	Header records checksum to this value.

The fields of the header structure are as follows:

c_type	The type of the header.
c_date	The date the dump was taken.
c_ddate	The date the file system was dumped from.
c_volume	The current volume number of the dump.
c_tapea	The current number of this (1024-byte) record.
c_inumber	The number of the inode being dumped if this is of type TS_INODE.
c_magic	This contains the value MAGIC above, truncated as needed.
c_checksum	This contains whatever value is needed to make the record sum to CHECKSUM.
c_dinode	This is a copy of the inode as it appears on the file system; see *fs*(5).
c_count	The count of characters in *c_addr*.
c_addr	An array of characters describing the blocks of the dumped file. A character is zero if the block associated with that character was not present on the file system, otherwise the character is non-zero. If the block was not present on the file system, no block was dumped; the block will be restored as a hole in the file. If there is not sufficient space in this record to describe all of the blocks in a file, TS_ADDR records will be scattered through the file, each one picking up where the last left off.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a TS_END record and then the tapemark.

The structure *idates* describes an entry in the file */etc/dumpdates* where dump history is kept. The fields of the structure are:

id_name	The dumped filesystem is '/dev/*id_nam*'.
id_incno	The level number of the dump tape; see *dump*(8).
id_ddate	The date of the incremental dump in system format see *types*(5).

FILES

/etc/dumpdates

SEE ALSO

dump(8), restore(8), fs(5), types(5)

BUGS

Should more explicitly describe format of dumpdates file.

NAME

environ – user environment

SYNOPSIS

extern char **environ;

DESCRIPTION

An array of strings called the 'environment' is made available by *execve*(2) when a process begins. By convention these strings have the form '*name=value*'. The following names are used by various commands:

PATH — The sequence of directory prefixes that *sh, time, nice*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by ':'. The *login*(1) process sets PATH=:/usr/ucb:/bin:/usr/bin.

HOME — A user's login directory, set by *login*(1) from the password file *passwd*(5).

TERM — The kind of terminal for which output is to be prepared. This information is used by commands, such as *nroff* or *plot*(1G), which may exploit special terminal capabilities. See */etc/termcap* (*termcap*(5)) for a list of terminal types.

SHELL — The file name of the user's login shell.

TERMCAP — The string describing the terminal in TERM, or the name of the termcap file, see *termcap*(3),*termcap*(5),

EXINIT — A startup list of commands read by *ex*(1), *edit*(1), and *vi*(1).

USER — The login name of the user.

Further names may be placed in the environment by the *export* command and 'name=value' arguments in *sh*(1), or by the *setenv* command if you use *csh*(1). Arguments may also be placed in the environment at the point of an *execve*(2). It is unwise to conflict with certain *sh*(1) variables that are frequently exported by '.profile' files: MAIL, PS1, PS2, IFS.

SEE ALSO

csh(1), ex(1), login(1), sh(1), getenv(3), execve(2), system(3), termcap(3X), termcap(5)

NAME

fcntl – file control options

DESCRIPTION

#include <fcntl.h>

DESCRIPTION

The *fcntl*(2) function provides for control over open files. This include file describes *requests* and *arguments* to *fcntl* and *open*(2) as shown below:

```
/*      @(#)fcntl.h 1.2 83/12/08 SMI; from UCB 4.2 83/09/25   */

/*
 * Flag values accessible to open(2) and fcntl(2)
 *  (The first three can only be set by open)
 */
#define O_RDONLY    0
#define O_WRONLY    1
#define O_RDWR              2
#define O_NDELAY    FNDELAY        /* Non-blocking I/O */
#define O_APPEND    FAPPEND        /* append (writes guaranteed at the end) */

#ifndef F_DUPFD
/* fcntl(2) requests */
#define F_DUPFD     0       /* Duplicate fildes */
#define F_GETFD     1       /* Get fildes flags */
#define F_SETFD     2       /* Set fildes flags */
#define F_GETFL     3       /* Get file flags */
#define F_SETFL     4       /* Set file flags */
#define F_GETOWN 5 /* Get owner */
#define F_SETOWN 6 /* Set owner */

/* flags for F_GETFL, F_SETFL-- copied from <sys/file.h> */
#define FNDELAY             00004          /* non-blocking reads */
#define FAPPEND             00010          /* append on each write */
#define FASYNC              00100          /* signal pgrp when data ready */
#endif
```

SEE ALSO

fcntl(2), open(2)

NAME

fs, inode – format of file system volume

SYNOPSIS

#include <sys/types.h>
#include <sys/filsys.h>
#include <sys/inode.h>

DESCRIPTION

Every file system storage volume (disk, nine-track tape, for instance) has a common format for certain vital information. Every such volume is divided into a certain number of blocks. The block size is a parameter of the file system. Sectors 0 to 15 on a file system are used to contain primary and secondary bootstrapping programs.

The actual file system begins at sector 16 with the *super block.* The layout of the super block as defined by the include file *<sys/fs.h>* is:

```
#define FS_MAGIC      0x011954
struct   fs {
         struct  fs *fs_link;         /* linked list of file systems */
         struct  fs *fs_rlink;        /*     used for incore super blocks */
         daddr_t fs_sblkno;           /* addr of super-block in filesys */
         daddr_t fs_cblkno;           /* offset of cyl-block in filesys */
         daddr_t fs_iblkno;           /* offset of inode-blocks in filesys */
         daddr_t fs_dblkno;           /* offset of first data after cg */
         long    fs_cgoffset;         /* cylinder group offset in cylinder */
         long    fs_cgmask;           /* used to calc mod fs_ntrak */
         time_t  fs_time;             /* last time written */
         long    fs_size;             /* number of blocks in fs */
         long    fs_dsize;            /* number of data blocks in fs */
         long    fs_ncg;              /* number of cylinder groups */
         long    fs_bsize;            /* size of basic blocks in fs */
         long    fs_fsize;            /* size of frag blocks in fs */
         long    fs_frag;             /* number of frags in a block in fs */
/* these are configuration parameters */
         long    fs_minfree;          /* minimum percentage of free blocks */
         long    fs_rotdelay;         /* num of ms for optimal next block */
         long    fs_rps;              /* disk revolutions per second */
/* these fields can be computed from the others */
         long    fs_bmask;            /* "blkoff" calc of blk offsets */
         long    fs_fmask;            /* "fragoff" calc of frag offsets */
         long    fs_bshift;           /* "lblkno" calc of logical blkno */
         long    fs_fshift;           /* "numfrags" calc number of frags */
/* these are configuration parameters */
         long    fs_maxcontig;        /* max number of contiguous blks */
         long    fs_maxbpg;           /* max number of blks per cyl group */
/* these fields can be computed from the others */
         long    fs_fragshift;        /* block to frag shift */
         long    fs_fsbtodb;          /* fsbtodb and dbtofsb shift constant */
         long    fs_sbsize;           /* actual size of super block */
         long    fs_csmask;           /* csum block offset */
         long    fs_csshift;          /* csum block number */
         long    fs_nindir;           /* value of NINDIR */
         long    fs_inopb;            /* value of INOPB */
         long    fs_nspf;             /* value of NSPF */
         long    fs_sparecon[6];      /* reserved for future constants */
```

```
/* sizes determined by number of cylinder groups and their sizes */
        daddr_t fs_csaddr;          /* blk addr of cyl grp summary area */
        long    fs_cssize;          /* size of cyl grp summary area */
        long    fs_cgsize;          /* cylinder group size */
/* these fields should be derived from the hardware */
        long    fs_ntrak;           /* tracks per cylinder */
        long    fs_nsect;           /* sectors per track */
        long    fs_spc;             /* sectors per cylinder */
/* this comes from the disk driver partitioning */
        long    fs_ncyl;            /* cylinders in file system */
/* these fields can be computed from the others */
        long    fs_cpg;             /* cylinders per group */
        long    fs_ipg;             /* inodes per group */
        long    fs_fpg;             /* blocks per group * fs_frag */
/* this data must be re-computed after crashes */
        struct  csum fs_cstotal; /* cylinder summary information */
/* these fields are cleared at mount time */
        char    fs_fmod;            /* super block modified flag */
        char    fs_clean;           /* file system is clean flag */
        char    fs_ronly;           /* mounted read-only flag */
        char    fs_flags;           /* currently unused flag */
        char    fs_fsmnt[MAXMNTLEN];        /* name mounted on */
/* these fields retain the current block allocation info */
        long    fs_cgrotor;         /* last cg searched */
        struct  csum *fs_csp[MAXCSBUFS];/* list of fs_cs info buffers */
        long    fs_cpc;             /* cyl per cycle in postbl */
        short   fs_postbl[MAXCPG][NRPOS];/* head of blocks for each rotation */
        long    fs_magic;           /* magic number */
        u_char  fs_rotbl[1];        /* list of blocks for each rotation */
/* actually longer */
};
```

Each disk drive contains some number of file systems. A file system consists of a number of cylinder groups. Each cylinder group has inodes and data.

A file system is described by its super-block, which in turn describes the cylinder groups. The super-block is critical data and is replicated in each cylinder group to protect against catastrophic loss. This is done at file system creation time and the critical super-block data does not change, so the copies need not be referenced further unless disaster strikes.

Addresses stored in inodes are capable of addressing fragments of 'blocks'. File system blocks of at most size MAXBSIZE can be optionally broken into 2, 4, or 8 pieces, each of which is addressable; these pieces may be DEV_BSIZE, or some multiple of a DEV_BSIZE unit.

Large files consist of exclusively large data blocks. To avoid undue wasted disk space, the last data block of a small file is allocated as only as many fragments of a large block as are necessary. The file system format retains only a single pointer to such a fragment, which is a piece of a single large block that has been divided. The size of such a fragment is determinable from information in the inode, using the "blksize(fs, ip, lbn)" macro.

The file system records space availability at the fragment level; to determine block availability, aligned fragments are examined.

The root inode is the root of the file system. Inode 0 can't be used for normal purposes and historically bad blocks were linked to inode 1, thus the root inode is 2 (inode 1 is no longer used for this purpose, however numerous dump tapes make this assumption, so we are stuck with it). The *lost+found* directory is given the next available inode when it is initially created by *mkfs*.

fs_minfree gives the minimum acceptable percentage of file system blocks which may be free. If the freelist drops below this level only the super-user may continue to allocate blocks. This may be set to 0 if no reserve of free blocks is deemed necessary, however severe performance degradations will be observed if the file system is run at greater than 90% full; thus the default value of *fs_minfree* is 10%.

Empirically the best trade-off between block fragmentation and overall disk utilization at a loading of 90% comes with a fragmentation of 4, thus the default fragment size is a fourth of the block size.

Cylinder group related limits: Each cylinder keeps track of the availability of blocks at different rotational positions, so that sequential blocks can be laid out with minimum rotational latency. NRPOS is the number of rotational positions which are distinguished. With NRPOS 8 the resolution of the summary information is 2ms for a typical 3600 rpm drive.

fs_rotdelay gives the minimum number of milliseconds to initiate another disk transfer on the same cylinder. It is used in determining the rotationally optimal layout for disk blocks within a file; the default value for *fs_rotdelay* is 2ms.

Each file system has a statically allocated number of inodes. An inode is allocated for each NBPI bytes of disk space. The inode allocation strategy is extremely conservative.

MAXIPG bounds the number of inodes per cylinder group, and is needed only to keep the structure simpler by having the only a single variable size element (the free bit map).

N.B.: MAXIPG must be a multiple of INOPB(fs).

MINBSIZE is the smallest allowable block size. With a MINBSIZE of 4096 it is possible to create files of size 2^32 with only two levels of indirection. MINBSIZE must be big enough to hold a cylinder group block, thus changes to (struct cg) must keep its size within MINBSIZE. MAXCPG is limited only to dimension an array in (struct cg); it can be made larger as long as that structure's size remains within the bounds dictated by MINBSIZE. Note that super blocks are never more than size SBSIZE.

The path name on which the file system is mounted is maintained in *fs_fsmnt*. MAXMNTLEN defines the amount of space allocated in the super block for this name. The limit on the amount of summary information per file system is defined by MAXCSBUFS. It is currently parameterized for a maximum of two million cylinders.

Per cylinder group information is summarized in blocks allocated from the first cylinder group's data blocks. These blocks are read in from *fs_csaddr* (size *fs_cssize*) in addition to the super block.

N.B.: sizeof (struct csum) must be a power of two in order for the "fs_cs" macro to work.

Super block for a file system: MAXBPC bounds the size of the rotational layout tables and is limited by the fact that the super block is of size SBSIZE. The size of these tables is **inversely** proportional to the block size of the file system. The size of the tables is increased when sector sizes are not powers of two, as this increases the number of cylinders included before the rotational pattern repeats (*fs_cpc*). The size of the rotational layout tables is derived from the number of bytes remaining in (struct fs).

MAXBPG bounds the number of blocks of data per cylinder group, and is limited by the fact that cylinder groups are at most one block. The size of the free block table is derived from the size of blocks and the number of remaining bytes in the cylinder group structure (struct cg).

Inode: The inode is the focus of all file activity in the UNIX file system. There is a unique inode allocated for each active file, each current directory, each mounted-on file, text file, and the root. An inode is 'named' by its device/i-number pair. For further information, see the include file <sys/inode.h>.

NAME

fstab – static information about the filesystems

SYNOPSIS

#include <fstab.h>

DESCRIPTION

The file */etc/fstab* describes the file systems and swapping partitions on the local machine. It is created by the system administrator using a text editor and processed by commands which mount, unmount, check consistency of, dump and restore file systems, and by the system in providing swap space.

It consists of a number of lines of the form:

```
fs_spec:fs_file:fs_type:fs_freq:fs_passno
```

an example of which would be:

```
/dev/xy0a:/:rw:1:1
```

The entries from this file are accessed using the routines in *getfsent*(3), which returns a structure of the following form:

```
struct fstab {
        char  *fs_spec;  /* block special device name */
        char  *fs_file;  /* file system path prefix */
        char  *fs_type;  /* rw,ro,sw or xx */
        int   fs_freq;   /* dump frequency, in days */
        int   fs_passno; /* pass number on parallel dump */
};
```

The lines in the file give for each file system or swap area on the local machine the disk partition it is contained in *fs_spec* and the directory on which it is to be mounted (unless it is a swap area) in *fs_file*. The *fs_spec* special file name is the **block** special file name, and not the character special file name which the rest of the entry refers to. If a program needs the character special file name, the program must create it by appending a "r" after the last "/" in the special file name.

The *fs_type* indicates whether it it to be read-only "ro", readable and writable "rw", or readable and writable subject to quotas "rq". If *fs_type* is "sw" then the special file is made available as a piece of swap space by the *swapon*(8) command at the end of the system reboot procedure. The fields other than *fs_spec* and *fs_type* are not used in this case. If *fs_type* is "rq" then at boot time the file system is automatically processed by the *quotacheck*(8) command and disk quotas are then enabled with *quotaon*(8). File system quotas are maintained in a file "quotas", which is located at the root of the associated file system. If *fs_type* is specified as "xx" the entry is ignored. This is useful to show disk partitions which are currently not used.

The field *fs_freq* indicates how often each partition should be dumped by the *dump*(8) command (and triggers that commands **w** option which tells which file systems should be dumped). Most systems set the *fs_freq* field to 1 indicating that the file systems are dumped each day.

The final field *fs_passno* is used by the disk consistency check program *fsck*(8) to allow overapped checking of file systems during a reboot. All file systems with *fs_passno* of 1 are first checked simultaneosly, then all file systems with *fs_passno* of 2, and so on. It is usual to make the *fs_passno* of the root file system have the value 1 and then check one file system on each available disk drive in each subsequent pass to the exhaustion of file system partitions.

/etc/fstab is only *read* by programs, and not written; it is the duty of the system administrator to properly create and maintain this file. The order of records in */etc/fstab* is important because *fsck, mount,* and *umount* process the file sequentially; file systems must appear *after* file systems they are mounted within.

FILES

/etc/fstab

SEE ALSO

getfsent(3), quotacheck(8), quotaon(8)

NAME

gettytab – terminal configuration data base

SYNOPSIS

/etc/gettytab

DESCRIPTION

Gettytab is a simplified version of the *termcap*(5) data base used to describe terminal lines. The initial terminal login process *getty*(8) accesses the *gettytab* file each time it starts, allowing simpler reconfiguration of terminal characteristics. Each entry in the data base is used to describe one class of terminals.

There is a default terminal class, *default*, that is used to set global defaults for all other classes. (That is, the *default* entry is read, then the entry for the class required is used to override particular settings.)

CAPABILITIES

Refer to *termcap*(5) for a description of the file layout. The *default* column below lists defaults obtained if there is no entry in the table obtained, nor one in the special *default* table.

Name	Type	Default	Description
ap	bool	false	terminal uses any parity
bd	num	0	backspace delay
bk	str	0377	alternate end of line character (input break)
cb	bool	false	use crt backspace mode
cd	num	0	carriage-return delay
ce	bool	false	use crt erase algorithm
ck	bool	false	use crt kill algorithm
cl	str	NULL	screen clear sequence
co	bool	false	console - add \n after login prompt
ds	str	^Y	delayed suspend character
ec	bool	false	leave echo OFF
ep	bool	false	terminal uses even parity
er	str	^?	erase character
et	str	^D	end of text (EOF) character
ev	str	NULL	initial enviroment
f0	num	unused	tty mode flags to write messages
f1	num	unused	tty mode flags to read login name
f2	num	unused	tty mode flags to leave terminal as
fd	num	0	form-feed (vertical motion) delay
fl	str	^O	output flush character
hc	bool	false	do NOT hangup line on last close
he	str	NULL	hostname editing string
hn	str	hostname	hostname
ht	bool	false	terminal has real tabs
ig	bool	false	ignore garbage characters in login name
im	str	NULL	initial (banner) message
in	str	^C	interrupt character
is	num	unused	input speed
kl	str	^U	kill character
lc	bool	false	terminal has lower case
lm	str	login:	login prompt
ln	str	^V	"literal next" character
lo	str	/bin/login	program to exec when name obtained
nd	num	0	newline (line-feed) delay
nl	bool	false	terminal has (or might have) a newline character

nx	str		default next table (for auto speed selection)
op	bool	false	terminal uses odd parity
os	num	unused	output speed
pc	str	\0	pad character
pe	bool	false	use printer (hard copy) erase algorithm
pf	num	0	delay between first prompt and following flush (seconds)
ps	bool	false	line connected to a MICOM port selector
qu	str	^\	quit character
rp	str	^R	line retype character
rw	bool	false	do NOT use raw for input, use cbreak
sp	num	unused	line speed (input and output)
su	str	^Z	suspend character
tc	str	none	table continuation
to	num	0	timeout (seconds)
tt	str	NULL	terminal type (for enviroment)
ub	bool	false	do unbuffered output (of prompts etc)
uc	bool	false	terminal is known upper case only
we	str	^W	word erase character
xc	bool	false	do NOT echo control chars as ^X
xf	str	^S	XOFF (stop output) character
xn	str	^Q	XON (start output) character

If no line speed is specified, speed will not be altered from that which prevails when *getty* is entered. Specifying an input or output speed overrides line speed for stated direction only.

Terminal modes to be used for the output of the message, for input of the login name, and to leave the terminal set as upon completion, are derived from the Boolean flags specified. If the derivation should prove inadequate, any (or all) of these three may be overriden with one of the **f0**, **f1**, or **f2** numeric specifications, which can be used to specify (usually in octal, with a leading '0') the exact values of the flags. Local (new tty) flags are set in the top 16 bits of this (32 bit) value.

Should *getty* receive a null character (presumed to indicate a line break) it will restart using the table indicated by the **nx** entry. If there is none, it will re-use its original table.

Delays are specified in milliseconds, the nearest possible delay available in the tty driver will be used. Should greater certainty be desired, delays with values 0, 1, 2, and 3 are interpreted as choosing that particular delay algorithm from the driver.

The **cl** screen clear string may be preceded by a (decimal) number of milliseconds of delay required (a la termcap). This delay is simulated by repeated use of the pad character **pc**.

The initial message, and login message, **im** and **lm** may include the character sequence **%h** to obtain the hostname. (**%%** obtains a single '%' character.) The hostname is normally obtained from the system, but may be set by the **hn** table entry. In either case it may be edited with **he**. The **he** string is a sequence of characters, each character that is neither '@' nor '#' is copied into the final hostname. A '@' in the **he** string, causes one character from the real hostname to be copied to the final hostname. A '#' in the **he** string, causes the next character of the real hostname to be skipped. Surplus '@' and '#' characters are ignored.

When *getty* execs the login process, given in the **lo** string (usually "/bin/login"), it will have set the enviroment to include the terminal type, as indicated by the **tt** string (if it exists). The **ev** string, can be used to enter additional data into the environment. It is a list of comma separated strings, each of which will presumably be of the form *name=value*.

If a non-zero timeout is specified, with **to**, then *getty* will exit within the indicated number of seconds, either having received a login name and passed control to *login*, or having received an alarm signal, and exited. This may be useful to hangup dial in lines.

Output from *getty* is even parity unless **op** is specified. **Op** may be specified with **ap** to allow any parity on input, but generate odd parity output. Note: this only applies while *getty* is being run, terminal driver limitations prevent a more complete implementation. *Getty* does not check parity of input characters in *RAW* mode.

SEE ALSO

termcap(5), getty(8).

NAME

group – group file

DESCRIPTION

Group contains for each group the following information:

group name
encrypted password
numerical group ID
a comma separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; Each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

FILES

/etc/group

SEE ALSO

setgroups(2), initgroups(3), crypt(3), passwd(1), passwd(5)

BUGS

The *passwd*(1) command won't change the passwords.

NAME

hosts – host name data base

DESCRIPTION

The *hosts* file contains information regarding the known hosts on the DARPA Internet. For each host a single line should be present with the following information:

official host name
Internet address
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official host data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown hosts.

Network addresses are specified in the conventional "." notation using the *inet_addr*() routine from the Internet address manipulation library, *inet*(3N). Host names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/hosts

SEE ALSO

gethostent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

NAME

kbd – keyboard translation table format and default table

SYNOPSIS

#include <sundev/kbd.h>

DESCRIPTION

Keyboard translation is done in the UNIX kernel via a set of tables. A translation table is 128 bytes of 'entries', which are bytes (unsigned chars). The top 4 bits of each entry are decoded by a case statement in the keyboard translator. If the entry is less than 0x80, it is sent out as an ASCII character (possibly with the META bit OR-ed in). 'Special' entries are 0x80 or greater, and invoke more complicated actions.

```
struct keymap {
        unsigned char   keymap[128];    /* maps keycodes to actions */
};
```

A keyboard is defined by its keymaps.

```
struct keyboard {
        struct keymap   *k_normal;      /* Unshifted */
        struct keymap   *k_shifted;     /* Shifted */
        struct keymap   *k_caps;        /* Caps locked */
        struct keymap   *k_control;     /* Controlled */
        struct keymap   *k_up;          /* Key went up */
        int             k_idleshifts;   /* Shifts */
        int             k_idlebuckys;   /* Bucky bits */
        unsigned char   k_abort1;       /* 1st key of abort sequence */
        unsigned char   k_abort2;       /* 2nd key of abort sequence */
};
```

The following defines the bit positions used within k_idleshifts to indicate the 'pressed' (1) or 'released' (0) state of shift keys. The bit numbers and the aggregate masks are defined.

Since it is possible to have more than one bit in the shift mask on at once, there is an implied priority given to each shift state when determining which translation table to use. The order is (from highest priority to lowest) UPMASK, CTRLMASK, SHIFTMASK, and lastly CAPSMASK.

```
#define CAPSLOCK     0        /* Caps Lock key */
#define SHIFTLOCK    1        /* Shift Lock key */
#define LEFTSHIFT    2        /* Left-hand shift key */
#define RIGHTSHIFT   3        /* Right-hand shift key */
#define LEFTCTRL     4        /* Left-hand (or only) control key */
#define RIGHTCTRL    5        /* Right-hand control key */
#define CAPSMASK     0x0001   /* Caplock translation table */
#define SHIFTMASK    0x000E   /* Shifted translation table */
#define CTRLMASK     0x0030   /* Ctrl shift translation table */
#define UPMASK       0x0080   /* Key up translation table */
```

Special Entry Keys

The 'special' entries' top 4 bits are defined below. Generally they are used with a 4-bit parameter (such as a bit number) in the low 4 bits. The bytes whose top 4 bits are 0x0 thru 0x7 happen to be ASCII characters. They are not special cased, but just normal cased.

```
#define SHIFTKEYS    0x80
```

thru 0x8F. This key helps to determine the translation table used. The bit position of its bit in 'shiftmask' is added to the entry, for example, SHIFTKEYS+ LEFTCTRL. When this entry is invoked, the bit in 'shiftmask' is toggled. Depending which tables you put it in, this works well for hold-down keys or press-on, press-off keys.

#define BUCKYBITS 0x90

thru 0x9F. This key determines the state of one of the 'bucky' bits above the returned ASCII character. This is basically a way to pass mode-key-up/down information back to the caller with each 'real' key depressed. The concept, and name 'bucky' (derivation unknown) comes from the MIT/SAIL 'TV' system — they had TOP, META, CTRL, and a few other bucky bits. The bit position of its bit in 'buckybits', minus 7, is added to the entry; for example, bit 0x00000400 is BUCKYBITS+ 3. The '-7' prevents us from messing up the ASCII char, and gives us 16 useful bucky bits. When this entry is invoked, the designated bit in 'buckybits' is toggled. Depending which tables you put it in, this works well for hold-down keys or press-on, press-off keys.

#define METABIT 0

Meta key depressed with key. This is the only user accessible bucky bit. This value is added to BUCKYBITS in the translation table.

#define SYSTEMBIT 1

'System' key was down w/key. This is a kernel-accessible bucky bit. This value is added to BUCKYBITS in the translation table. The system key is currently not used except as a place holder to indicate the key used as the *k_abort1* key (as defined above).

```
#define FUNNY       0xA0  /* thru 0xAF. This key does one of 16 funny
                             things based on the low 4 bits: */
#define NOP         0xA0  /* This key does nothing. */
#define OOPS        0xA1  /* This key exists but is undefined. */
#define HOLE        0xA2  /* This key does not exist on the keyboard.
                             Its position code should never be
                             generated. This indicates a software/
                             hardware mismatch, or bugs. */
#define NOSCROLL    0xA3  /* This key alternately sends ^S or ^Q */
#define CTRLS       0xA4  /* This sends ^S and lets NOSCROLL know */
#define CTRLQ       0xA5  /* This sends ^Q and lets NOSCROLL know */
#define RESET       0xA6  /* Kbd was just reset */
#define ERROR       0xA7  /* Kbd just detected an internal error */
#define IDLE        0xA8  /* Kbd is idle (no keys down) */
```

Combinations 0xA9 to 0xAF are reserved for non-parameterized functions.

#define STRING 0xB0

thru 0xBF. The low-order 4 bits index a table select a string to be returned, char by char. Each entry in the table is null terminated.

```
#define KTAB_STRLEN     10    /* Maximum string length (including null) */
```

Definitions for the individual string numbers:

```
#define HOMEARROW       0x00
#define UPARROW         0x01
#define DOWNARROW       0x02
#define LEFTARROW       0x03
#define RIGHTARROW      0x04
```

String numbers 5 thru F are available to users making custom entries.

Function Key Groupings

In the following function key groupings, the low-order 4 bits indicate the function key number within the group:

```
#define LEFTFUNC      0xC0   /* thru 0xCF. The 'left' group. */
#define RIGHTFUNC     0xD0   /* thru 0xDF. The 'right' group. */
#define TOPFUNC       0xE0   /* thru 0xEF. The 'top' group. */
#define BOTTOMFUNC    0xF0   /* thru 0xFF. The 'bottom' group. */
#define LF(n)         (LEFTFUNC+(n)-1)
#define RF(n)         (RIGHTFUNC+(n)-1)
#define TF(n)         (TOPFUNC+(n)-1)
#define BF(n)         (BOTTOMFUNC+(n)-1)
```

The actual keyboard positions may not be on the left/right/top/bottom of the physical keyboard (although they usually are). What is important is that we have reserved 64 keys for function keys.

Normally, when a function key is pressed, the following escape sequence is sent through the character stream:

ESC[0..9z

where ESC is a single escape character and 0..9 indicate some number of digits needed to encode the function key as a decimal number.

DEFAULT TABLES

The kernel has 3 sets of initial translation tables, one set for each type of keyboard supported.

```
#ifndef lint
static  char sccsid[] = "@(#)keytables.c 1.3 83/10/25 Copyr 1983 Sun Micro";
#endif

/*
 * Copyright (C) 1983 by Sun Microsystems, Inc.
 */

/*
 * keytables.c
 *
 * This module contains the translation tables for the up-down encoded
 * Sun keyboards.
 */
#include "../sun/kbd.h"

/* handy way to define control characters in the tables */
#define c(char) (char&0x1F)
#define ESC 0x1B


/* Unshifted keyboard table for Micro Switch 103SD32-2 */

static struct keymap keytab_ms_lc = {
/* 0 */HOLE, BUCKYBITS+SYSTEMBIT,
                        LF(2),  LF(3),  HOLE,  TF(1),  TF(2),  TF(3),
/* 8 */TF(4),  TF(5),  TF(6),  TF(7),  TF(8),  TF(9),  TF(10), TF(11),
/* 16 */       TF(12), TF(13), TF(14),c('['),  HOLE,  RF(1),  '+',    '-',
```

```
/* 24 */        HOLE, LF(4),  '\f',    LF(6),  HOLE, SHIFTKEYS+ CAPSLOCK,
                                                             '1',    '2',
/* 32 */        '3',   '4',   '5',     '6',    '7',   '8',   '9',    '0',
/* 40 */        '-',   '~',   '`',     '\b',   HOLE, '7',    '8',    '9',
/* 48 */        HOLE, LF(7),  STRING+ UPARROW,
                                       LF(9),  HOLE, '\t',   'q',    'w',
/* 56 */        'e',   'r',   't',     'y',    'u',   'i',   'o',    'p',
/* 64 */        '{',   '}',   '_',     HOLE,  '4',   '5',   '6',    HOLE,
/* 72 */        STRING+ LEFTARROW,
                       STRING+ HOMEARROW,
                              STRING+ RIGHTARROW,
                                       HOLE, SHIFTKEYS+ SHIFTLOCK,
                                                     'a',   's',    'd',
/* 80 */        'f',   'g',   'h',     'j',    'k',   'l',   ';',    ':',
/* 88 */        '|',   '\r',  HOLE,   '1',    '2',   '3',   HOLE, NOSCROLL,
/* 96 */        STRING+ DOWNARROW,
                       LF(97), HOLE, HOLE, SHIFTKEYS+ LEFTSHIFT,
                                                     'z',   'x',    'c',
/*104 */        'v',   'b',   'n',     'm',    ',',   '.',   '/',    SHIFTKEYS+ RIGHTSHIFT,
/*112 */        NOP,   0x7F,  '0',     NOP,    '.',   HOLE, HOLE, HOLE,
/*120 */        HOLE, HOLE, SHIFTKEYS+ LEFTCTRL,
                                       ' ',    SHIFTKEYS+ RIGHTCTRL,
                                                      HOLE, HOLE, IDLE,
};

/* Shifted keyboard table for Micro Switch 103SD32-2 */

static struct keymap keytab_ms_uc = {
/*  0 */HOLE, BUCKYBITS+ SYSTEMBIT,
                              LF(2),   LF(3),  HOLE, TF(1),  TF(2),  TF(3),
/*  8 */TF(4),  TF(5),  TF(6),  TF(7),  TF(8),  TF(9),  TF(10), TF(11),
/* 16 */        TF(12), TF(13), TF(14),c('['), HOLE, RF(1), '+',    '-',
/* 24 */        HOLE, LF(4),  '\f',    LF(6),  HOLE, SHIFTKEYS+ CAPSLOCK,
                                                             '!',    '"',
/* 32 */        '#',   '$',   '%',     '&',    '\'',  '(',   ')',    '0',
/* 40 */        '=',   '^',   '@',     '\b',   HOLE, '7',    '8',    '9',
/* 48 */        HOLE, LF(7),  STRING+ UPARROW,
                                       LF(9),  HOLE, '\t',   'Q',    'W',
/* 56 */        'E',   'R',   'T',     'Y',    'U',   'I',   'O',    'P',
/* 64 */        '[',   ']',   '_',     HOLE,  '4',   '5',   '6',    HOLE,
/* 72 */        STRING+ LEFTARROW,
                       STRING+ HOMEARROW,
                              STRING+ RIGHTARROW,
                                       HOLE, SHIFTKEYS+ SHIFTLOCK,
                                                     'A',   'S',    'D',
/* 80 */        'F',   'G',   'H',     'J',    'K',   'L',   '+',    '*',
/* 88 */        '\\',  '\r',  HOLE,   '1',    '2',   '3',   HOLE, NOSCROLL,
/* 96 */        STRING+ DOWNARROW,
                       LF(97), HOLE, HOLE, SHIFTKEYS+ LEFTSHIFT,
                                                     'Z',   'X',    'C',
/*104 */        'V',   'B',   'N',     'M',    '<',   '>',   '?',    SHIFTKEYS+ RIGHTSHIFT,
/*112 */        NOP,   0x7F,  '0',     NOP,    '.',   HOLE, HOLE, HOLE,
/*120 */        HOLE, HOLE, SHIFTKEYS+ LEFTCTRL,
```

```
                                    ' ',      SHIFTKEYS+ RIGHTCTRL,
                                              HOLE, HOLE, IDLE,
};

/* Caps Locked keyboard table for Micro Switch 103SD32-2 */

static struct keymap keytab_ms_cl = {
/*  0 */HOLE, BUCKYBITS+ SYSTEMBIT,
                          LF(2),    LF(3),   HOLE,  TF(1),   TF(2),   TF(3),
/*  8 */TF(4), TF(5),  TF(6),   TF(7),    TF(8),   TF(9),  TF(10),  TF(11),
/* 16 */       TF(12), TF(13),  TF(14),c('|'),     HOLE,   RF(1),   '+',     '-',
/* 24 */       HOLE,   LF(4),   '\f',     LF(6),   HOLE,   SHIFTKEYS+ CAPSLOCK,
                                                                    '1',     '2',
/* 32 */       '3',    '4',     '5',      '6',     '7',    '8',     '9',     '0',
/* 40 */       '-',    '~',     '`',      '\b',    HOLE,   '7',     '8',     '9',
/* 48 */       HOLE,   LF(7),   STRING+ UPARROW,
                                          LF(9),   HOLE,   '\t',    'Q',     'W',
/* 56 */       'E',    'R',     'T',      'Y',     'U',    'I',     'O',     'P',
/* 64 */       '{',    '}',     '_',      HOLE,    '4',    '5',     '6',     HOLE,
/* 72 */       STRING+ LEFTARROW,
                       STRING+ HOMEARROW,
                                STRING+ RIGHTARROW,
                                          HOLE,    SHIFTKEYS+ SHIFTLOCK,
                                                           'A',     'S',     'D',
/* 80 */       'F',    'G',     'H',      'J',     'K',    'L',     ';',     ':',
/* 88 */       '|',    '\r',    HOLE,     '1',     '2',    '3',     HOLE,    NOSCROLL,
/* 96 */       STRING+ DOWNARROW,
                       LF(97),  HOLE,     HOLE,    SHIFTKEYS+ LEFTSHIFT,
                                                           'Z',     'X',     'C',
/*104 */       'V',    'B',     'N',      'M',     ',',    '.',     '/',     SHIFTKEYS+ RIGHTSHIFT,
/*112 */       NOP,    0x7F,    '0',      NOP,     '.',    HOLE,    HOLE,    HOLE,
/*120 */       HOLE,   HOLE,    SHIFTKEYS+ LEFTCTRL,
                                          ' ',     SHIFTKEYS+ RIGHTCTRL,
                                                           HOLE,    HOLE,    IDLE,
};

/* Controlled keyboard table for Micro Switch 103SD32-2 */

static struct keymap keytab_ms_ct = {
/*  0 */HOLE, BUCKYBITS+ SYSTEMBIT,
                          LF(2),    LF(3),   HOLE,  TF(1),   TF(2),   TF(3),
/*  8 */TF(4), TF(5),  TF(6),   TF(7),    TF(8),   TF(9),  TF(10),  TF(11),
/* 16 */       TF(12), TF(13),  TF(14),c('|'),     HOLE,   RF(1),   OOPS,    OOPS,
/* 24 */       HOLE,   LF(4),   '\f',     LF(6),   HOLE,   SHIFTKEYS+ CAPSLOCK,
                                                                    OOPS,    OOPS,
/* 32 */       OOPS,   OOPS,    OOPS,     OOPS,    OOPS,   OOPS,    OOPS,    OOPS,
/* 40 */       OOPS,   c('^'),  c('@'),   '\b',    HOLE,   OOPS,    OOPS,    OOPS,
/* 48 */       HOLE,   LF(7),   STRING+ UPARROW,
                                          LF(9),   HOLE,   '\t',    CTRLQ,        c('W'),
/* 56 */       c('E'), c('R'),  c('T'),   c('Y'),  c('U'), c('I'),  c('O'),  c('P'),
/* 64 */       c('['), c(']'),  c('_'),   HOLE,    OOPS,   OOPS,    OOPS,    HOLE,
/* 72 */       STRING+ LEFTARROW,
```

```
                        STRING+ HOMEARROW,
                            STRING+ RIGHTARROW,
                                HOLE, SHIFTKEYS+ SHIFTLOCK,
                                        c('A'),  CTRLS,     c('D'),
/* 80 */    c('F'), c('G'), c('H'), c('J'), c('K'), c('L'), OOPS, OOPS,
/* 88 */    c('\\'),
                '\r',   HOLE, OOPS, OOPS, OOPS, HOLE, NOSCROLL,
/* 96 */    STRING+ DOWNARROW,
                LF(97), HOLE, HOLE, SHIFTKEYS+ LEFTSHIFT,
                                        c('Z'),  c('X'),  c('C'),
/*104 */    c('V'), c('B'), c('N'), c('M'), OOPS, OOPS, OOPS, SHIFTKEYS+ RIGHTSHIFT,
/*112 */    NOP,  0x7F,  OOPS, NOP,  OOPS, HOLE, HOLE, HOLE,
/*120 */    HOLE, HOLE, SHIFTKEYS+ LEFTCTRL,
                                '\0',   SHIFTKEYS+ RIGHTCTRL,
                                        HOLE, HOLE, IDLE,
};


/* "Key Up" keyboard table for Micro Switch 103SD32-2 */

static struct keymap keytab_ms_up = {
/*  0 */HOLE, BUCKYBITS+ SYSTEMBIT,
                        NOP,  NOP,  HOLE, NOP,  NOP,  NOP,
/*  8 */NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,
/* 16 */    NOP,  NOP,  NOP,  NOP,  HOLE, NOP,  NOP,  NOP,
/* 24 */    HOLE, NOP,  NOP,  NOP,  HOLE, SHIFTKEYS+ CAPSLOCK,
                                              NOP,  NOP,
/* 32 */    NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,
/* 40 */    NOP,  NOP,  NOP,  NOP,  HOLE, NOP,  NOP,  NOP,
/* 48 */    HOLE, NOP,  NOP,  NOP,  HOLE, NOP,  NOP,  NOP,
/* 56 */    NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,
/* 64 */    NOP,  NOP,  NOP,  HOLE, NOP,  NOP,  NOP,  HOLE,
/* 72 */    NOP,  NOP,  NOP,  HOLE, SHIFTKEYS+ SHIFTLOCK,
                                        NOP,  NOP,  NOP,
/* 80 */    NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,
/* 88 */    NOP,  NOP,  HOLE, NOP,  NOP,  NOP,  HOLE, NOP,
/* 96 */    NOP,  NOP,  HOLE, HOLE, SHIFTKEYS+ LEFTSHIFT,
                                        NOP,  NOP,  NOP,
/*104 */    NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  SHIFTKEYS+ RIGHTSHIFT,
/*112 */    NOP,  NOP,  NOP,  NOP,  NOP,  HOLE, HOLE, HOLE,
/*120 */    HOLE, HOLE, SHIFTKEYS+ LEFTCTRL,
                                NOP,  SHIFTKEYS+ RIGHTCTRL,
                                        HOLE, HOLE, RESET,
};


/* Index to keymaps for Micro Switch 103SD32-2 */
static struct keyboard keyindex_ms = {
        &keytab_ms_lc,
        &keytab_ms_uc,
        &keytab_ms_cl,
        &keytab_ms_ct,
        &keytab_ms_up,
```

```
        CTLSMASK,     /* Shift bits which stay on with idle keyboard */
        0x0000,       /* Bucky bits which stay on with idle keyboard */
        1,      77,   /* abort keys */
};

/* Unshifted keyboard table for Sun-2 keyboard */

static struct keymap keytab_s2_lc = {
/*  0 */HOLE,  BUCKYBITS+ SYSTEMBIT,
                             LF(2),  LF(3),  HOLE,  TF(1),  TF(2),  TF(3),
/*  8 */TF(4),  TF(5),  TF(6),  TF(7),  TF(8),  TF(9),  TF(10), TF(11),
/* 16 */        TF(12), TF(13), TF(14), TF(15), HOLE,  RF(1),  RF(2),  RF(3),
/* 24 */        HOLE,  LF(4),  LF(5),  LF(6),  HOLE,  c('['),  '1',    '2',
/* 32 */        '3',    '4',    '5',    '6',    '7',    '8',    '9',    '0',
/* 40 */        '-',    '=',    '`',    '\b',   HOLE,  RF(4),  RF(5),  RF(6),
/* 48 */        HOLE,  LF(7),  LF(8),  LF(9),  HOLE,  '\t',   'q',    'w',
/* 56 */        'e',    'r',    't',    'y',    'u',    'i',    'o',    'p',
/* 64 */        '[',    ']',    0x7F,   HOLE,  RF(7),  STRING+ UPARROW,
                                                             RF(9),  HOLE,
/* 72 */        LF(10), LF(11), LF(12), HOLE,  SHIFTKEYS+ LEFTCTRL,
                                                     'a',    's',    'd',
/* 80 */        'f',    'g',    'h',    'j',    'k',    'l',    ';',    '\'',
/* 88 */        '\\',   '\r',   HOLE,  STRING+ LEFTARROW,
                                             RF(11), STRING+ RIGHTARROW,
                                                             HOLE,  LF(13),
/* 96 */        LF(14), LF(15), HOLE,  SHIFTKEYS+ LEFTSHIFT,
                                             'z',    'x',    'c',    'v',
/*104 */        'b',    'n',    'm',    ',',    '.',    '/',    SHIFTKEYS+ RIGHTSHIFT,
                                                                     '\n',
/*112 */        RF(13), STRING+ DOWNARROW,
                             RF(15), HOLE,  HOLE,  HOLE,  HOLE,  HOLE,
/*120 */        BUCKYBITS+ METABIT,
                     ' ',    BUCKYBITS+ METABIT,
                                     HOLE,  HOLE,  HOLE,  ERROR,       IDLE,
};

/* Shifted keyboard table for Sun-2 keyboard */

static struct keymap keytab_s2_uc = {
/*  0 */HOLE,  BUCKYBITS+ SYSTEMBIT,
                             LF(2),  LF(3),  HOLE,  TF(1),  TF(2),  TF(3),
/*  8 */TF(4),  TF(5),  TF(6),  TF(7),  TF(8),  TF(9),  TF(10), TF(11),
/* 16 */        TF(12), TF(13), TF(14), TF(15), HOLE,  RF(1),  RF(2),  RF(3),
/* 24 */        HOLE,  LF(4),  LF(5),  LF(6),  HOLE,  c('['),  '!',    '@',
/* 32 */        '#',    '$',    '%',    '^',    '&',    '*',    '(',    ')',
/* 40 */        '_',    '+',    '~',    '\b',   HOLE,  RF(4),  RF(5),  RF(6),
/* 48 */        HOLE,  LF(7),  LF(8),  LF(9),  HOLE,  '\t',   'Q',    'W',
/* 56 */        'E',    'R',    'T',    'Y',    'U',    'I',    'O',    'P',
/* 64 */        '{',    '}',    0x7F,   HOLE,  RF(7),  STRING+ UPARROW,
                                                             RF(9),  HOLE,
/* 72 */        LF(10), LF(11), LF(12), HOLE,  SHIFTKEYS+ LEFTCTRL,
                                                     'A',    'S',    'D',
/* 80 */        'F',    'G',    'H',    'J',    'K',    'L',    ':',    '"',
```

```
/* 88 */        '|',     '\r',    HOLE, STRING+ LEFTARROW,
                                        RF(11), STRING+ RIGHTARROW,
                                                    HOLE, LF(13),
/* 96 */        LF(14), LF(15), HOLE, SHIFTKEYS+ LEFTSHIFT,
                                        'Z',     'X',     'C',     'V',
/*104 */        'B',     'N',     'M',     '<',     '>',     '?',     SHIFTKEYS+ RIGHTSHIFT,
                                                             '\n',
/*112 */        RF(13), STRING+ DOWNARROW,
                                RF(15), HOLE, HOLE, HOLE, HOLE, HOLE,
/*120 */        BUCKYBITS+ METABIT,
                        ' ',     BUCKYBITS+ METABIT,
                                        HOLE, HOLE, HOLE, ERROR,        IDLE,
};
```

```
/* Controlled keyboard table for Sun-2 keyboard */

static struct keymap keytab_s2_ct = {
/*  0 */HOLE, BUCKYBITS+ SYSTEMBIT,
                        LF(2),   LF(3),   HOLE,    TF(1),   TF(2),   TF(3),
/*  8 */TF(4),  TF(5),   TF(6),   TF(7),   TF(8),   TF(9),   TF(10),  TF(11),
/* 16 */        TF(12),  TF(13),  TF(14),  TF(15),  HOLE,    RF(1),   RF(2),   RF(3),
/* 24 */        HOLE,    LF(4),   LF(5),   LF(6),   HOLE,    c('|'),  '1',     c('@'),
/* 32 */        '3',     '4',     '5',     c('^'),  '7',     '8',     '9',     '0',
/* 40 */        c('_'),  '=',     c('^'),  '\b',    HOLE,    RF(4),   RF(5),   RF(6),
/* 48 */        HOLE,    LF(7),   LF(8),   LF(9),   HOLE,    '\t',    c('q'),  c('w'),
/* 56 */        c('e'),  c('r'),  c('t'),  c('y'),  c('u'),  c('i'),  c('o'),  c('p'),
/* 64 */        c('['),  c(']'),  0x7F,    HOLE,    RF(7),   STRING+ UPARROW,
                                                             RF(9),   HOLE,
/* 72 */        LF(10),  LF(11),  LF(12),  HOLE,    SHIFTKEYS+ LEFTCTRL,
                                                    c('a'),  c('s'),  c('d'),
/* 80 */        c('f'),  c('g'),  c('h'),  c('j'),  c('k'),  c('l'),  ';',     '\'',
/* 88 */        c('\\'),
                        '\r',    HOLE,    STRING+ LEFTARROW,
                                          RF(11), STRING+ RIGHTARROW,
                                                             HOLE,    LF(13),
/* 96 */        LF(14),  LF(15),  HOLE,    SHIFTKEYS+ LEFTSHIFT,
                                          c('z'),  c('x'),  c('c'),  c('v'),
/*104 */        c('b'),  c('n'),  c('m'),  ',',     '.',     c('_'),  SHIFTKEYS+ RIGHTSHIFT,
                                                                      '\n',
/*112 */        RF(13),  STRING+ DOWNARROW,
                                 RF(15), HOLE, HOLE, HOLE, HOLE, HOLE,
/*120 */        BUCKYBITS+ METABIT,
                        c(' '),  BUCKYBITS+ METABIT,
                                          HOLE, HOLE, HOLE, ERROR,        IDLE,
};
```

```
/* "Key Up" keyboard table for Sun-2 keyboard */

static struct keymap keytab_s2_up = {
/*  0 */HOLE, BUCKYBITS+ SYSTEMBIT,
```

```
                        OOPS, OOPS, HOLE, OOPS, OOPS, OOPS,
/*  8 */OOPS, OOPS, OOPS, OOPS, OOPS, OOPS, OOPS, OOPS,
/* 16 */        OOPS, OOPS, OOPS, OOPS, HOLE, OOPS, OOPS, NOP,
/* 24 */        HOLE, OOPS, OOPS, OOPS, HOLE, NOP,  NOP,  NOP,
/* 32 */        NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,
/* 40 */        NOP,  NOP,  NOP,  NOP,  HOLE, OOPS, OOPS, NOP,
/* 48 */        HOLE, OOPS, OOPS, OOPS, HOLE, NOP,  NOP,  NOP,
/* 56 */        NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,
/* 64 */        NOP,  NOP,  NOP,  HOLE, OOPS, OOPS, NOP,  HOLE,
/* 72 */        OOPS, OOPS, OOPS, HOLE, SHIFTKEYS+LEFTCTRL,
                                              NOP,  NOP,  NOP,
/* 80 */        NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,
/* 88 */        NOP,  NOP,  HOLE, OOPS, OOPS, NOP,  HOLE, OOPS,
/* 96 */        OOPS, OOPS, HOLE, SHIFTKEYS+LEFTSHIFT,
                                        NOP,  NOP,  NOP,  NOP,
/*104 */        NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  SHIFTKEYS+RIGHTSHIFT,
                                                          NOP,
/*112 */        OOPS, OOPS, NOP,  HOLE, HOLE, HOLE, HOLE, HOLE,
/*120 */        BUCKYBITS+METABIT,
                      NOP,  BUCKYBITS+METABIT,
                                  HOLE, HOLE, HOLE, HOLE, RESET,
};

/* Index to keymaps for Sun-2 keyboard */
static struct keyboard keyindex_s2 = {
        &keytab_s2_lc,
        &keytab_s2_uc,
        0,
        &keytab_s2_ct,
        &keytab_s2_up,
        0x0000,         /* Shift bits which stay on with idle keyboard */
        0x0000,         /* Bucky bits which stay on with idle keyboard */
        1,      77,     /* abort keys */
};

/* Unshifted keyboard table for "VT100 style" */

static struct keymap keytab_vt_lc = {
/*  0 */HOLE, BUCKYBITS+SYSTEMBIT,
                        HOLE, HOLE, HOLE, HOLE, HOLE, HOLE,
/*  8 */HOLE, HOLE, STRING+UPARROW,
                              STRING+DOWNARROW,
                                    STRING+LEFTARROW,
                                          STRING+RIGHTARROW,
                                                HOLE, TF(1),
/* 16 */        TF(2), TF(3), TF(4), c('['), '1',  '2',  '3',  '4',
/* 24 */        '5',   '6',   '7',   '8',    '9',  '0',  '-',  '=',
/* 32 */        '`',   c('H'), BUCKYBITS+METABIT,
                                     '7',    '8',  '9',  '-',  '\t',
/* 40 */        'q',   'w',   'e',   'r',    't',  'y',  'u',  'i',
/* 48 */        'o',   'p',   '[',   ']',    0x7F, '4',  '5',  '6',
/* 56 */        ',',   SHIFTKEYS+LEFTCTRL,
                       SHIFTKEYS+CAPSLOCK,
```

```
                                        'a',    's',    'd',    'f',    'g',
/* 64 */        'h',    'j',    'k',    'l',    ';',    '\'',   '\r',   '\\',
/* 72 */        '1',    '2',    '3',    NOP,    NOSCROLL,
                                                        SHIFTKEYS+LEFTSHIFT,
                                                                'z',    'x',
/* 80 */        'c',    'v',    'b',    'n',    'm',    ',',    '.',    '/',
/* 88 */        SHIFTKEYS+RIGHTSHIFT,
                        '\n',   '0',    HOLE,   '.',    '\r',   HOLE,   HOLE,
/* 96 */        HOLE,   HOLE,   ' ',    HOLE,   HOLE,   HOLE,   HOLE,   HOLE,
/*104 */        HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,
/*112 */        HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,
/*120 */        HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   IDLE,
};

/* Shifted keyboard table for "VT100 style" */

static struct keymap keytab_vt_uc = {
/*  0 */HOLE,   BUCKYBITS+SYSTEMBIT,
                                HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,
/*  8 */HOLE,   HOLE,   STRING+UPARROW,
                                        STRING+DOWNARROW,
                                                STRING+LEFTARROW,
                                                        STRING+RIGHTARROW,
                                                                HOLE,   TF(1),
/* 16 */        TF(2),  TF(3),  TF(4),  c('['), '!',    '@',    '#',    '$',
/* 24 */        '%',    '^',    '&',    '*',    '(',    ')',    '_',    '+',
/* 32 */        '~',    c('H'), BUCKYBITS+METABIT,
                                        '7',    '8',    '9',    '-',    '\t',
/* 40 */        'Q',    'W',    'E',    'R',    'T',    'Y',    'U',    'I',
/* 48 */        'O',    'P',    '{',    '}',    0x7F,   '4',    '5',    '6',
/* 56 */        ',',    SHIFTKEYS+LEFTCTRL,
                                SHIFTKEYS+CAPSLOCK,
                                        'A',    'S',    'D',    'F',    'G',
/* 64 */        'H',    'J',    'K',    'L',    ':',    '"',    '\r',   '|',
/* 72 */        '1',    '2',    '3',    NOP,    NOSCROLL,
                                                        SHIFTKEYS+LEFTSHIFT,
                                                                'Z',    'X',
/* 80 */        'C',    'V',    'B',    'N',    'M',    '<',    '>',    '?',
/* 88 */        SHIFTKEYS+RIGHTSHIFT,
                        '\n',   '0',    HOLE,   '.',    '\r',   HOLE,   HOLE,
/* 96 */        HOLE,   HOLE,   ' ',    HOLE,   HOLE,   HOLE,   HOLE,   HOLE,
/*104 */        HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,
/*112 */        HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,
/*120 */        HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   IDLE,
};

/* Caps Locked keyboard table for "VT100 style" */

static struct keymap keytab_vt_cl = {
/*  0 */HOLE,   BUCKYBITS+SYSTEMBIT,
                                HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,
/*  8 */HOLE,   HOLE,   STRING+UPARROW,
```

```
                                        STRING+DOWNARROW,
                                                STRING+LEFTARROW,
                                                        STRING+RIGHTARROW,
                                                                HOLE, TF(1),
/* 16 */        TF(2),  TF(3),  TF(4),  c('|'), '1',    '2',    '3',    '4',
/* 24 */        '5',    '6',    '7',    '8',    '9',    '0',    '-',    '=',
/* 32 */        '`',    c('H'), BUCKYBITS+METABIT,
                                        '7',    '8',    '9',    '-',    '\t',
/* 40 */        'Q',    'W',    'E',    'R',    'T',    'Y',    'U',    'I',
/* 48 */        'O',    'P',    '[',    ']',    0x7F,   '4',    '5',    '6',
/* 56 */        ',',    SHIFTKEYS+LEFTCTRL,
                                SHIFTKEYS+CAPSLOCK,
                                        'A',    'S',    'D',    'F',    'G',
/* 64 */        'H',    'J',    'K',    'L',    ';',    '\'',   '\r',   '\\',
/* 72 */        '1',    '2',    '3',    NOP,    NOSCROLL,
                                                        SHIFTKEYS+LEFTSHIFT,
                                                                'Z',    'X',
/* 80 */        'C',    'V',    'B',    'N',    'M',    ',',    '.',    '/',
/* 88 */        SHIFTKEYS+RIGHTSHIFT,
                        '\n',   '0',    HOLE,   '.',    '\r',   HOLE,   HOLE,
/* 96 */        HOLE,   HOLE,   ' ',    HOLE,   HOLE,   HOLE,   HOLE,   HOLE,
/*104 */        HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,
/*112 */        HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,
/*120 */        HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   IDLE,
};

/* Controlled keyboard table for "VT100 style" */

static struct keymap keytab_vt_ct = {
/*  0 */HOLE, BUCKYBITS+SYSTEMBIT,
                                HOLE,   HOLE,   HOLE,   HOLE,   HOLE,   HOLE,
/*  8 */HOLE, HOLE, STRING+UPARROW,
                                        STRING+DOWNARROW,
                                                STRING+LEFTARROW,
                                                        STRING+RIGHTARROW,
                                                                HOLE, TF(1),
/* 16 */        TF(2),  TF(3),  TF(4),  c('|'), '1',    c('@'), '3',    '4',
/* 24 */        '5',    c('^'), '7',    '8',    '9',    '0',    c('_'), '=',
/* 32 */        c('^'), c('H'), BUCKYBITS+METABIT,
                                        '7',    '8',    '9',    '-',    '\t',
/* 40 */        CTRLQ,          c('W'), c('E'), c('R'), c('T'), c('Y'), c('U'), c('I'),
/* 48 */        c('O'), c('P'), c('['), c(']'), 0x7F,   '4',    '5',    '6',
/* 56 */        ',',    SHIFTKEYS+LEFTCTRL,
                                SHIFTKEYS+CAPSLOCK,
                                        c('A'), CTRLS,          c('D'), c('F'), c('G'),
/* 64 */        c('H'), c('J'), c('K'), c('L'), ':',    '"',    '\r',   c('\\'),
/* 72 */        '1',    '2';    '3',    NOP,    NOSCROLL,
                                                        SHIFTKEYS+LEFTSHIFT,
                                                                c('Z'), c('X'),
/* 80 */        c('C'), c('V'), c('B'), c('N'), c('M'), ',',    '.',    c('_'),
/* 88 */        SHIFTKEYS+RIGHTSHIFT,
                        '\n',   '0',    HOLE,   '.',    '\r',   HOLE,   HOLE,
/* 96 */        HOLE,   HOLE,   c(' '), HOLE,   HOLE,   HOLE,   HOLE,   HOLE,
```

```
/*104 */      HOLE, HOLE, HOLE, HOLE, HOLE, HOLE, HOLE, HOLE,
/*112 */      HOLE, HOLE, HOLE, HOLE, HOLE, HOLE, HOLE, HOLE,
/*120 */      HOLE, HOLE, HOLE, HOLE, HOLE, HOLE, HOLE, IDLE,
};

/* "Key up" keyboard table for "VT100 style" */

static struct keymap keytab_vt_up = {
/*  0 */HOLE, BUCKYBITS+SYSTEMBIT,
                          HOLE, HOLE, HOLE, HOLE, HOLE, HOLE,
/*  8 */HOLE, HOLE, NOP,  NOP,  NOP,  NOP,  HOLE, NOP,
/* 16 */      NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,
/* 24 */      NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,
/* 32 */      NOP,  NOP,  BUCKYBITS+METABIT,
                                NOP,  NOP,  NOP,  NOP,  NOP,
/* 40 */      NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,
/* 48 */      NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,
/* 56 */      NOP,  SHIFTKEYS+LEFTCTRL,
                    SHIFTKEYS+CAPSLOCK,
                                NOP,  NOP,  NOP,  NOP,  NOP,
/* 64 */      NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,
/* 72 */      NOP,  NOP,  NOP,  NOP,  NOP,  SHIFTKEYS+LEFTSHIFT,
                                            NOP,  NOP,
/* 80 */      NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,  NOP,
/* 88 */      SHIFTKEYS+RIGHTSHIFT,
                    NOP,  NOP,  HOLE, NOP,  NOP,  HOLE, HOLE,
/* 96 */      HOLE, HOLE, NOP,  HOLE, HOLE, HOLE, HOLE, HOLE,
/*104 */      HOLE, HOLE, HOLE, HOLE, HOLE, HOLE, HOLE, HOLE,
/*112 */      HOLE, HOLE, HOLE, HOLE, HOLE, HOLE, HOLE, HOLE,
/*120 */      HOLE, HOLE, HOLE, HOLE, HOLE, HOLE, HOLE, RESET,
};

/* Index to keymaps for "VT100 style" keyboard */
static struct keyboard keyindex_vt = {
        &keytab_vt_lc,
        &keytab_vt_uc,
        &keytab_vt_cl,
        &keytab_vt_ct,
        &keytab_vt_up,
        CAPSMASK+CTLSMASK,      /* Shift keys that stay on at idle keyboard */
        0x0000,         /* Bucky bits that stay on at idle keyboard */
        1,      59,     /* abort keys */
};

/***************************************************************************/
/*  Index table for the whole shebang                                      */
/***************************************************************************/
int nkeytables = 3;     /* max 16 */
struct keyboard *keytables[] = {
        &keyindex_ms,
        &keyindex_vt,
```

```
        &keyindex_s2,
};

/*
        Keyboard String Table

        This defines the strings sent by various keys (as selected in the
        tables above).
*/

#define kstescinit(c)   {'\033', '[', 'c', '\0'}
char keystringtab[16][KTAB_STRLEN] = {
        kstescinit(H) /*home*/,
        kstescinit(A) /*up*/,
        kstescinit(B) /*down*/,
        kstescinit(D) /*left*/,
        kstescinit(C) /*right*/,
};
```

SEE ALSO

cons(4S)

BUGS

This keyboard translation implementation is essentially the PROM monitor mechanism moved into the kernel. It will almost certainly be reworked in the future to take advantage of the greater flexibility available to the kernel that was not available in the PROM.

NAME

mtab – mounted file system table

SYNOPSIS

#include <fstab.h>
#include <mtab.h>

DESCRIPTION

Mtab resides in directory */etc* and contains a table of devices mounted by the *mount* command. *Umount* removes entries.

The table is a series of *mtab* structures, as defined in <mtab.h>. Each entry contains the null-padded name of the place where the special file is mounted, the null-padded name of the special file, and a type field, one of those defined in *<fstab.h>*. The special file has all its directories stripped away; that is, everything through the last '/' is thrown away. The type field indicates if the file system is mounted read-only, read-write, or read-write with disk quotas enabled.

This table is present only so people can look at it. It does not matter to *mount* if there are duplicated entries nor to *umount* if a name cannot be found.

FILES

/etc/mtab

SEE ALSO

mount(8)

NAME

networks – network name data base

DESCRIPTION

The *networks* file contains information regarding the known networks which comprise the DARPA Internet. For each network a single line should be present with the following information:

official network name
network number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official network data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown networks.

Network number may be specified in the conventional "." notation using the *inet_network*() routine from the Internet address manipulation library, *inet*(3N). Network names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/networks

SEE ALSO

getnetent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

NAME

news – USENET network news article, utility files

DESCRIPTION

There are two formats of news articles: A and B. A format is the only format that version 1 netnews systems can read or write. Systems running the version 2 netnews can read either format and there are provisions for the version 2 netnews to write in A format. A format looks like this:

A*article-ID*
newsgroups
path
date
title
Body of article

Only version 2 netnews systems can read and write B format. B format contains two extra pieces of information: receival date and expiration date. The basic structure of a B format file consists of a series of headers and then the body. A header field is defined as a line with a capital letter in the 1st column and a colon somewhere on the line. Unrecognized header fields are ignored. News is stored in the same format transmitted, see "Standard for the Interchange of USENET Messages" for a full description. The following fields are among those recognized:

Header	Information
From:	*user@host.domain[.domain ...] (Full Name)*
Newsgroups:	*Newsgroups*
Message-ID:	*<Unique Identifier>*
Subject:	*descriptive title*
Date:	*Date Posted*
Date-Received:	*Date received on local machine*
Expires:	*Expiration Date*
Reply-To:	*Address for mail replies*
References:	*Article ID of article this is*
Control:	*Text of a control message*

Here is an example of an article:

```
Relay-Version: B 2.10   2/13/83 cbosgd.UUCP
Posting-Version: B 2.10  2/13/83 eagle.UUCP
Path: cbosgd!mhuxj!mhuxt!eagle!jerry
From: jerry@eagle.uucp (Jerry Schwarz)
Newsgroups: net.general
Subject: Usenet Etiquette -- Please Read
Message-ID: <642@eagle.UUCP>
Date: Friday, 19-Nov-82 16:14:55 EST
Followup-To: net.news
Expires: Saturday, 1-Jan-83 00:00:00 EST
Date-Received: Friday, 19-Nov-82 16:59:30 EST
Organization: Bell Labs, Murray Hill

The body of the article comes here, after a blank line.
```

A *sys* file line has four fields, each seperated by colons:

system-name:subscriptions:flags:transmission command

Of these fields, on the *system-name* and *subscriptions* need to be present.

The *system name* is the name of the system being sent to. The *subscriptions* is the list of newsgroups to be transmitted to the system. The *flags* are a set of letters describing how the article should be transmitted. The default is B. Valid flags include A (send in A format), B (send in B format), N (use ihave/sendme protocol), U (use uux -c and the name of the stored article in a %s string).

The *transmission command* is executed by the shell with the article to be transmitted as the standard input. The default is **uux - -z -r** *sysname***!rnews.** Some examples:

xyz:net.all
oldsys:net.all,fa.all,to.oldsys:A
berksys:net.all,ucb.all::/usr/lib/news/sendnews -b berksysrnews
arpasys:net.all,arpa.all::/usr/lib/news/sendnews -a rnews@arpasys
old2:net.all,fa.all:A:/usr/lib/sendnews -o old2rnews
user:fa.sf-lovers::mail user

Somewhere in a *sys* file, there must be a line for the host system. This line has no *flags* or *commands*. A # as the first character in a line denotes a comment.

The history, active, and ngfile files have one line per item.

SEE ALSO

inews(1), postnews(1), sendnews(8), uurec(8), readnews(1)

NAME

newsrc – information file for readnews(1) and checknews(1)

DESCRIPTION

The *.newsrc* file contains the list of previously read articles and an optional options line for *readnews*(1) and *checknews*(1). Each newsgroup that articles have been read from has a line of the form:

.IR newsgroup : " range"

Range is a list of the articles read. It is basically a list of numbers separated by commas with sequential numbers collapsed with hyphens. For instance:

general: 1-78,80,85-90
fa.info-cpm: 1-7
net.news: 1
fa.info-vax! 1-5

If the : is replaced with an ! (as in info-vax above) the newsgroup is not subscribed to and is not be shown to the user.

An options line starts with the word **options** (left-justified). Then there are the list of options just as they would be on the command line. For instance:

options –n all !fa.sf-lovers !fa.human-nets –r
options –c –r

A string of lines beginning with a space or tab after the initial options line are considered continuation lines.

FILES

~/.newsrc options and list of previously read articles

SEE ALSO

readnews(1), checknews(1)

NAME

passwd – password file

DESCRIPTION

Passwd contains for each user the following information:

name (login name, contains no upper case)
encrypted password
numerical user ID
numerical group ID
user's real name, office, extension, home phone.
initial working directory
program to use as Shell

The name may contain '&', meaning insert the login name.

The password file is an ASCII file. Each field within each user's entry is separated from the next by a colon. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, */bin/sh* is used.

The password file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user ID's to names.

Appropriate precautions must be taken to lock the file against changes if it is to be edited with a text editor; *vipw*(8) does the necessary locking.

FILES

/etc/passwd

SEE ALSO

getpwent(3), login(1), crypt(3), passwd(1), group(5), vipw(8), adduser(8)

BUGS

A binary indexed file format should be available for fast access.

User information (name, office, etc.) should be stored elsewhere.

NAME

phones – remote host phone number data base

DESCRIPTION

The file /etc/phones contains the system-wide private phone numbers for the *tip*(1C) program. This file is normally unreadable, and so may contain privileged information. The format of the file is a series of lines of the form: <system-name>[\t]*<phone-number>. The system name is one of those defined in the *remote*(5) file and the phone number is constructed from [0123456789-=*%]. The "=" and "*" characters are indicators to the auto call units to pause and wait for a second dial tone (when going through an exchange). The "=" is required by the DF02-AC and the "*" is required by the BIZCOMP 1030.

Only one phone number per line is permitted. However, if more than one line in the file contains the same system name *tip*(1C) will attempt to dial each one in turn, until it establishes a connection.

FILES

/etc/phones

SEE ALSO

tip(1C), remote(5)

NAME

plot – graphics interface

DESCRIPTION

Files of this format are produced by routines described in *plot*(3X), and are interpreted for various devices by commands described in *plot*(1G). A graphics file is a stream of plotting instructions. Each instruction consists of an ASCII letter usually followed by bytes of binary information. The instructions are executed in order. A point is designated by four bytes representing the x and y values; each value is a signed integer. The last designated point in an **l, m, n,** or **p** instruction becomes the 'current point' for the next instruction.

Each of the following descriptions begins with the name of the corresponding routine in *plot*(3X).

m move: The next four bytes give a new current point.

n cont: Draw a line from the current point to the point given by the next four bytes. See *plot*(1G).

p point: Plot the point given by the next four bytes.

l line: Draw a line from the point given by the next four bytes to the point given by the following four bytes.

t label: Place the following ASCII string so that its first character falls on the current point. The string is terminated by a newline.

a arc: The first four bytes give the center, the next four give the starting point, and the last four give the end point of a circular arc. The least significant coordinate of the end point is used only to determine the quadrant. The arc is drawn counter-clockwise.

c circle: The first four bytes give the center of the circle, the next two the radius.

e erase: Start another frame of output.

f linemod: Take the following string, up to a newline, as the style for drawing further lines. The styles are 'dotted,' 'solid,' 'longdashed,' 'shortdashed,' and 'dotdashed.' Effective only in *plot 4014* and *plot ver*.

s space: The next four bytes give the lower left corner of the plotting area; the following four give the upper right corner. The plot will be magnified or reduced to fit the device as closely as possible.

Space settings that exactly fill the plotting area with unity scaling appear below for devices supported by the filters of *plot*(1G). The upper limit is just outside the plotting area. In every case the plotting area is taken to be square; points outside may be displayable on devices whose face isn't square.

4014	space(0, 0, 3120, 3120);
ver	space(0, 0, 2048, 2048);
300, 300s	space(0, 0, 4096, 4096);
450	space(0, 0, 4096, 4096);

SEE ALSO

plot(1G), plot(3X), graph(1G)

NAME

printcap – printer capability data base

SYNOPSIS

/etc/printcap

DESCRIPTION

Printcap is a simplified version of the *termcap*(5) data base for describing printers. The spooling system accesses the *printcap* file every time it is used, allowing dynamic addition and deletion of printers. Each entry in the data base describes one printer. This data base may not be substituted for, as is possible for *termcap*, because it may allow accounting to be bypassed.

The default printer is normally *lp*, though the environment variable *PRINTER* may be used to override this. Each spooling utility supports a –P*printer* option to explicitly name a destination printer.

Refer to the *Line Printer Spooler Manual* in the Sun *System Manager's Manual* for a discussion of how to set up the database for a given printer.

Each entry in the *printcap* file describes a printer, and is a line consisting of a number of fields separated by ':' characters. The first entry for each printer gives the names which are known for the printer, separated by '|' characters. The first name is conventionally a number. The second name given is the most common abbreviation for the printer, and the last name given should be a long name fully identifying the printer. The second name should contain no blanks; the last name may well contain blanks for readability. Entries may continue onto multiple lines by giving a \ as the last character of a line, and empty fields may be included for readability.

Capabilities in *printcap* are all introduced by two-character codes, and are of three types:

Boolean capabilities indicate that the printer has some particular feature. Boolean capabilities are simply written between the ':' characters, and are indicated by the word 'bool' in the **type** column of the capabilities table below.

Numeric capabilities supply information such as baud-rates, number of lines per page, and so on. Numeric capabilities are indicated by the word 'num' in the **type** column of the capabilities table below. Numeric capabilities are given by the two-character capability code followed by the '#' character, followed by the numeric value. For example: :br#1200: is a numeric entry stating that this printer should run at 1200 baud.

String capabilities give a sequence which can be used to perform particular printer operations such as cursor motion. String valued capabilities are indicated by the word 'str' in the **type** column of the capabilities table below. String valued capabilities are given by the two-character capability code followed by an '=' sign and then a string ending at the next following ':'. For example, :rp=spinwriter: is a sample entry stating that the remote printer is named 'spinwriter'.

CAPABILITIES

Name	**Type**	**Default**	**Description**
af	str	NULL	name of accounting file
br	num	none	if lp is a tty, set the baud rate (ioctl call)
cf	str	NULL	cifplot data filter
df	str	NULL	TeX data filter (DVI format)
du	str	0	User ID of user 'daemon'.
fc	num	0	if lp is a tty, clear flag bits (sgtty.h)
ff	str	"\f"	string to send for a form feed
fo	bool	false	print a form feed when device is opened
fs	num	0	like 'fc' but set bits
gf	str	NULL	graph data filter (plot (3X) format)
ic	bool	false	driver supports (non standard) ioctl

			call for indenting printout
if	str	NULL	name of text filter which does accounting
lf	str	"/dev/console"	error logging file name
lo	str	"lock"	name of lock file
lp	str	"/dev/lp"	device name to open for output
mc	num	0	maximum number of copies
mx	num	1000	maximum file size (in BUFSIZ blocks), zero = unlimited
nd	str	NULL	next directory for list of queues (unimplemented)
nf	str	NULL	ditroff data filter (device independent troff)
of	str	NULL	name of output filtering program
pl	num	66	page length (in lines)
pw	num	132	page width (in characters)
px	num	0	page width in pixels (horizontal)
py	num	0	page length in pixels (vertical)
rf	str	NULL	filter for printing FORTRAN style text files
rm	str	NULL	machine name for remote printer
rp	str	"lp"	remote printer name argument
rs	bool	false	restrict remote users to those with local accounts
rw	bool	false	open printer device read/write instead of read-only
sb	bool	false	short banner (one line only)
sc	bool	false	suppress multiple copies
sd	str	"/usr/spool/lpd"	spool directory
sf	bool	false	suppress form feeds
sh	bool	false	suppress printing of burst page header
st	str	"status"	status file name
tf	str	NULL	troff data filter (cat phototypesetter)
tr	str	NULL	trailer string to print when queue empties
vf	str	NULL	raster image filter
xc	num	0	if lp is a tty, clear local mode bits (tty (4))
xs	num	0	like 'xc' but set bits

Error messages sent to the console have a carriage return and a line feed appended to them, rather than just a line feed.

If the local line printer driver supports indentation, the daemon must understand how to invoke it.

SEE ALSO

termcap(5), lpc(8), lpd(8), pac(8), lpr(1), lpq(1), lprm(1)
The *Line Printer Spooler Manual* in the Sun *System Manager's Manual*.

NAME

protocols – protocol name data base

SYNOPSIS

/etc/protocols

DESCRIPTION

The *protocols* file contains information regarding the known protocols used in the DARPA Internet. For each protocol a single line should be present with the following information:

official protocol name
protocol number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Protocol names may contain any printable character other than a field delimiter, newline, or comment character.

EXAMPLE

The following example is taken from the Sun UNIX system.

```
#
# Internet (IP) protocols
#
ip      0   IP      # internet protocol, pseudo protocol number
icmp    1   ICMP    # internet control message protocol
ggp     2   GGP     # gateway-gateway protocol
tcp     6   TCP     # transmission control protocol
pup     12  PUP     # PARC universal packet protocol
udp     17  UDP     # user datagram protocol
```

FILES

/etc/protocols

SEE ALSO

getprotoent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

NAME

remote – remote host description file

DESCRIPTION

The systems known by *tip*(1C) and their attributes are stored in an ASCII file which is structured somewhat like the *termcap*(5) file. Each line in the file provides a description for a single *system*. Fields are separated by a colon (":"). Lines ending in a \ character with an immediately following newline are continued on the next line.

The first entry is the name(s) of the host system. If there is more than one name for a system, the names are separated by vertical bars. After the name of the system comes the fields of the description. A field name followed by an '=' sign indicates a string value follows. A field name followed by a '#' sign indicates a following numeric value.

Entries named "tip*" and "cu*" are used as default entries by *tip*, and the *cu* interface to *tip*, as follows. When *tip* is invoked with only a phone number, it looks for an entry of the form "tip300", where 300 is the baud rate with which the connection is to be made. When the *cu* interface is used, entries of the form "cu300" are used.

CAPABILITIES

Capabilities are either strings (str), numbers (num), or boolean flags (bool). A string capability is specified by *capability=value*; e.g. "dv=/dev/harris". A numeric capability is specified by *capability#value*; e.g. "xa#99". A boolean capability is specified by simply listing the capability.

at (str) Auto call unit type.

br (num) The baud rate used in establishing a connection to the remote host. This is a decimal number. The default baud rate is 300 baud.

cm (str) An initial connection message to be sent to the remote host. For example, if a host is reached through port selector, this might be set to the appropriate sequence required to switch to the host.

cu (str) Call unit if making a phone call. Default is the same as the 'dv' field.

dl (str) Disconnect message sent to the host when a disconnect is requested by the user.

du (bool) This host is on a dial-up line.

dv (str) UNIX device(s) to open to establish a connection. If this file refers to a terminal line, *tip*(1C) attempts to perform an exclusive open on the device to insure only one user at a time has access to the port.

el (str) Characters marking an end-of-line. The default is NULL. '~' escapes are only recognized by *tip* after one of the characters in 'el', or after a carriage-return.

fs (str) Frame size for transfers. The default frame size is equal to BUFSIZ.

hd (bool) The host uses half-duplex communication, local echo should be performed.

ie (str) Input end-of-file marks. The default is NULL.

oe (str) Output end-of-file string. The default is NULL. When *tip* is transferring a file, this string is sent at end-of-file.

pa (str) The type of parity to use when sending data to the host. This may be one of "even", "odd", "none", "zero" (always set bit 8 to zero), "one" (always set bit 8 to 1). The default is even parity.

pn (str) Telephone number(s) for this host. If the telephone number field contains an @ sign, *tip* searches the file */etc/phones* file for a list of telephone numbers; c.f. *phones*(5).

tc (str) Indicates that the list of capabilities is continued in the named description. This is used primarily to share common capability information.

Here is a short example showing the use of the capability continuation feature:

```
UNIX-1200:\
	:dv=/dev/cau0:el=^D^U^C^S^Q^O@:du:at=ventel:ie=#$%:oe=^D:br#1200:
arpavax|ax:\
	:pn=7654321%:tc=UNIX-1200
```

FILES

/etc/remote

SEE ALSO

tip(1C), phones(5)

NAME

sccsfile – format of SCCS file

DESCRIPTION

An SCCS file is an ASCII file. It consists of six logical parts: the *checksum*, the *delta table* (contains information about each delta), *user names* (contains login names and/or numerical group IDs of users who may add deltas), *flags* (contains definitions of internal keywords), *comments* (contains arbitrary descriptive information about the file), and the *body* (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the ASCII SOH (start of heading) character (octal 001). This character is hereafter referred to as *the control character* and will be represented graphically as **@**. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form DDDDD represent a five digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

```
@hDDDDD
```

The value of the checksum is the sum of all characters, except those of the first line. The **@h** provides a *magic number* of (octal) 064001.

Delta table

The delta table consists of a variable number of entries of the form:

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> DDDDD DDDDD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
 .
 .
 .
@c <comments> ...
 .
 .
 .
@e
```

The first line (**@s**) contains the number of lines inserted/deleted/unchanged respectively. The second line (**@d**) contains the type of the delta (currently, normal: D, and removed: R), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The **@i**, **@x**, and **@g** lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The **@m** lines (optional) each contain one MR number associated with the delta; the **@c** lines contain comments associated with the delta.

The **@e** line ends the delta table entry.

User names

The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines @u and @U. An empty list allows anyone to make a delta.

Flags

Keywords used internally (see *admin*(1) for more information on their use). Each flag line takes the form:

@f <flag> <optional text>

The following flags are defined:

@f t <type of program>
@f v <program name>
@f i
@f b
@f m <module name>
@f f <floor>
@f c <ceiling>
@f d <default-sid>
@f n
@f j
@f l <lock-releases>
@f q <user defined>

The t flag defines the replacement for the identification keyword. The v flag controls prompting for MR numbers in addition to comments; if the optional text is present it defines an MR number validity checking program. The i flag controls the warning/error aspect of the "No id keywords" message. When the i flag is not present, this message is only a warning; when the i flag is present, this message will cause a "fatal" error (the file will not be gotten, or the delta will not be made). When the b flag is present the -b keyletter may be used on the *get* command to cause a branch in the delta tree. The m flag defines the first choice for the replacement text of the sccsfile.5 identification keyword. The f flag defines the "floor" release; the release below which no deltas may be added. The c flag defines the "ceiling" release; the release above which no deltas may be added. The d flag defines the default SID to be used when none is specified on a *get* command. The n flag causes *delta* to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (for example, when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the n flag causes skipped releases to be completely empty. The j flag causes *get* to allow concurrent edits of the same base SID. The l flag defines a *list* of releases that are *locked* against editing (*get*(1) with the -e keyletter). The q flag defines the replacement for the %Q% identification keyword.

Comments

Arbitrary text surrounded by the bracketing lines @t and @T. The comments section typically will contain a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines don't begin with the control character, control lines do. There are three kinds of control lines: *insert*, *delete*, and *end*, represented by:

@I DDDDD

@D DDDDD
@E DDDDD

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

admin(1), delta(1), get(1), prs(1).
Source Code Control System User's Guide by L. E. Bonanni and C. A. Salemi.

NAME

servers – inet server data base

DESCRIPTION

The *servers* file contains the list of servers that *inetd*(8) operates. For each server a single line should be present with the following information:

name of server
protocol
server location

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

The name of the server should be the official service name as contained in *services*(5). The protocol entry is either udp or tcp. The server location is the full path name of the server program.

EXAMPLE

The following example is taken from the Sun UNIX system.

```
tcp        tcp  /usr/etc/in.tcpd
telnet     tcp  /usr/etc/in.telnetd
shell      tcp  /etc/in.rshd
login      tcp  /etc/in.rlogind
exec       tcp  /usr/etc/in.rexecd
ttcp       udp  /usr/etc/in.ttcpd
syslog     udp  /usr/etc/in.syslog
comsat     udp  /usr/etc/in.comsat
talk       udp  /usr/etc/in.talkd
time       tcp  /usr/etc/in.timed
```

FILES

/etc/servers

SEE ALSO

services(5), inetd(8)

NAME

services – service name data base

SYNOPSIS

/etc/services

DESCRIPTION

The *services* file contains information regarding the known services available in the DARPA Internet. For each service a single line should be present with the following information:

official service name
port number
protocol name
aliases

Items are separated by any number of blanks and/or tab characters. The port number and protocol name are considered a single *item*; a "/" is used to separate the port and protocol (for instance, "512/tcp"). A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Service names may contain any printable character other than a field delimiter, newline, or comment character.

EXAMPLE

Here is an example of the */etc/services* file from the Sun UNIX System.

```
#
# Network services, Internet style
#
echo            7/udp
discard         9/udp           sink null
systat          11/tcp
daytime         13/tcp
netstat         15/tcp
ftp             21/tcp
telnet          23/tcp
smtp            25/tcp          mail
time            37/tcp          timserver
name            42/tcp          nameserver
whois           43/tcp
mtp             57/tcp          # deprecated
#
# Host specific functions
#
tftp            69/udp
rje             77/tcp
finger          79/tcp
link            87/tcp          ttylink
supdup          95/tcp
#
# UNIX specific services
#
exec            512/tcp
login           513/tcp
shell           514/tcp         cmd
efs             520/tcp
biff            512/udp         comsat
who             513/udp         whod
```

syslog	514/udp	
talk	517/udp	
route	520/udp	router routed# 521 also

FILES

/etc/services

SEE ALSO

getservent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

NAME

tar – tape archive file format

DESCRIPTION

Tar, (the tape archive command) dumps several files into one, in a medium suitable for transportation.

A "tar tape" or file is a series of blocks. Each block is of size TBLOCK. A file on the tape is represented by a header block which describes the file, followed by zero or more blocks which give the contents of the file. At the end of the tape are two blocks filled with binary zeros, as an end-of-file indicator.

The blocks are grouped for physical I/O operations. Each group of *n* blocks (where *n* is set by the **b** keyletter on the *tar*(1) command line — default is 20 blocks) is written with a single system call; on nine-track tapes, the result of this write is a single tape record. The last group is always written at the full size, so blocks after the two zero blocks contain random data. On reading, the specified or default group size is used for the first read, but if that read returns less than a full tape block, the reduced block size is used for further reads, unless the **B** keyletter is used.

The header block looks like:

```
#define TBLOCK      512
#define NAMSIZ      100

union hblock {
        char dummy[TBLOCK];
        struct header {
                char name[NAMSIZ];
                char mode[8];
                char uid[8];
                char gid[8];
                char size[12];
                char mtime[12];
                char chksum[8];
                char linkflag;
                char linkname[NAMSIZ];
        } dbuf;
};
```

Name is a null-terminated string. The other fields are zero-filled octal numbers in ASCII. Each field (of width w) contains w-2 digits, a space, and a null, except *size* and *mtime*, which do not contain the trailing null. *Name* is the name of the file, as specified on the *tar* command line. Files dumped because they were in a directory which was named in the command line have the directory name as prefix and */filename* as suffix. *Mode* is the file mode, with the top bit masked off. *Uid* and *gid* are the user and group numbers which own the file. *Size* is the size of the file in bytes. Links and symbolic links are dumped with this field specified as zero. *Mtime* is the modification time of the file at the time it was dumped. *Chksum* is a decimal ASCII value which represents the sum of all the bytes in the header block. When calculating the checksum, the *chksum* field is treated as if it were all blanks. *Linkflag* is ASCII '0' if the file is "normal" or a special file, ASCII '1' if it is an hard link, and ASCII '2' if it is a symbolic link. The name linked-to, if any, is in *linkname,* with a trailing null. Unused fields of the header are binary zeros (and are included in the checksum).

The first time a given i-node number is dumped, it is dumped as a regular file. The second and subsequent times, it is dumped as a link instead. Upon retrieval, if a link entry is retrieved, but not the file it was linked to, an error message is printed and the tape must be manually re-scanned to retrieve the linked-to file.

The encoding of the header is designed to be portable across machines.

SEE ALSO

tar(1)

BUGS

Names or linknames longer than NAMSIZ produce error reports and cannot be dumped.

NAME

term – terminal driving tables for nroff

DESCRIPTION

Nroff(1) uses driving tables to customize its output for various types of output devices, such as printing terminals, special word-processing terminals (such as Diablo, Qume, or NEC Spinwriter mechanisms), or special output filter programs. These driving tables are written as C programs, compiled, and installed in **/usr/lib/term/tab***name* **, where** *name* is the name for that terminal type as given in *term*(7).

Here's how to construct a driver table for USG UNIX "nroff", in 25 easy lessons. The only changes for the V7 nroff (on 4.xBSD as well) are that the "iton" and "itoff" entries are missing, the "bset" and "breset" entries affect the "sg_flags" word in the "sgtty" structure, and the procedures for making the table are different.

Special thanks to the people at AT&T responsible for the UNIX documentation, without whose help this posting would not have been necessary. The structure of the tables is as follows:

```
#define    INCH      240

struct {
           int bset;
           int breset;
           int Hor;
           int Vert;
           int Newline;
           int Char;
           int Em;
           int Halfline;
           int Adj;
           char *twinit;
           char *twrest;
           char *twnl;
           char *hlr;
           char *hlf;
           char *flr;
           char *bdon;
           char *bdoff;
           char *iton;
           char *itoff;
           char *ploton;
           char *plotoff;
           char *up;
           char *down;
           char *right;
           char *left;
           char *codetab[256-32];
           char *zzz;
} t;
```

The meanings of the various fields are as follows:

bset bits to set in the *c_oflag* field of the *termio* structure (see *tty*(4)) before output.

breset bits to reset in the *c_oflag* field of the *termio* structure before output.

Hor horizontal resolution in fractions of an inch.

Vert vertical resolution in fractions of an inch.

Newline space moved by a newline (linefeed) character in fractions of an inch.

Char quantum of character sizes, in fractions of an inch. (that is, a character is a multiple of Char units wide)

Em size of an em in fractions of an inch.

Halfline space moved by a half-linefeed (or half-reverse-linefeed) character in fractions of an inch.

Adj quantum of white space, in fractions of an inch. (that is, white spaces are a multiple of Adj units wide)

Note: if this is less than the size of the space character (in units of Char; see below for how the sizes of characters are defined), *nroff* will output fractional spaces using plot mode. Also, if the –e switch to *nroff* is used, Adj is set equal to Hor by *nroff*.

twinit set of characters used to initialize the terminal in a mode suitable for *nroff*.

twrest set of characters used to restore the terminal to normal mode.

twnl set of characters used to move down one line.

hlr set of characters used to move up one-half line.

hlf set of characters used to move down one-half line.

flr set of characters used to move up one line.

bdon set of characters used to turn on hardware boldface mode, if any.

bdoff set of characters used to turn off hardware boldface mode, if any.

iton set of characters used to turn on hardware italics mode, if any.

itoff set of characters used to turn off hardware italics mode, if any.

ploton set of characters used to turn on hardware plot mode (for Diablo type mechanisms), if any.

plotoff set of characters used to turn off hardware plot mode (for Diablo type mechanisms), if any.

up set of characters used to move up one resolution unit (Vert) in plot mode, if any.

down set of characters used to move down one resolution unit (Vert) in plot mode, if any.

right set of characters used to move right one resolution unit (Hor) in plot mode, if any.

left set of characters used to move left one resolution unit (Hor) in plot mode, if any.

codetab definition of characters needed to print an *nroff* character on the terminal. The first byte is the number of character units (Char) needed to hold the character; that is, "\001" is one unit wide, "\002" is two units wide, etc. The high-order bit (0200) is on if the character is to be underlined in underline mode (.ul). The rest of the bytes are the characters used to produce the character in question. If the character has the sign (0200) bit on, it is a code to move the terminal in plot mode. It is encoded as:

0100 bit on	vertical motion.
0100 bit off	horizontal motion.
040 bit on	negative (up or left) motion.
040 bit off	positive (down or right) motion.
037 bits	number of such motions to make.

zzz a zero terminator at the end.

All quantities which are in units of fractions of an inch should be expressed as INCH**num/denom*, where *num* and *denom* are respectively the numerator and denominator of the fraction; that is, 1/48 of an inch would be written as "INCH/48".

If any sequence of characters does not pertain to the output device, that sequence should be given as a null string.

The source code for the terminal *name* is in **/usr/src/cmd/text/roff.d/terms.d/tab*name*.c.** When a new terminal type is added, the file *maketerms.c* should be updated to '#include' the source to that driving table; note that the various terminal types are grouped into "parts" labelled **PART1**, **PART2**, and **PART3**. If necessary, more parts can be added. Other changes necessary to *maketerms.c* are left as an exercise to the reader. The makefile *terms.mk* in that directory should then be updated.

FILES

/usr/lib/term/tab*name* driving tables
tab*name*.c source for driving tables

SEE ALSO

troff(1), term(7)

NAME

termcap – terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

Termcap is a data base describing terminals, used, for example, by *vi*(1) and *curses*(3X). Terminals are described in *termcap* by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in *termcap*.

Each entry in the *termcap* file describes a terminal, and is a line consisting of a number of fields separated by ':' characters. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name is always 2 characters long and is used by older version 6 systems which store the terminal type in a 16 bit word in a systemwide data base. The second name given is the most common abbreviation for the terminal, and the last name given should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may well contain blanks for readability. Entries may continue onto multiple lines by giving a \ as the last character of a line, and empty fields may be included for readability.

Capabilities in *termcap* are all introduced by two-character codes, and are of three types:

Boolean capabilities indicate that the terminal has some particular feature. Boolean capabilities are simply written between the ':' characters, and are indicated by the word 'bool' in the **type** column of the capabilities table below.

Numeric capabilities supply information such as the size of the terminal or the size of particular delays. Numeric capabilities are indicated by the word 'num' in the **type** column of the capabilities table below. Numeric capabilities are given by the two-character capability code followed by the '#' character and then the numeric value. For example: :co#80: is a numeric entry stating that this terminal has 80 columns.

String capabilities give a sequence which can be used to perform particular terminal operations such as cursor motion. String valued capabilities are indicated by the word 'str' in the **type** column of the capabilities table below. String valued capabilities are given by the two-character capability code followed by an '=' sign and then a string ending at the next following ':'. For example, :ce=16\E^S: is a sample entry for clear to end-of-line.

CAPABILITIES

(P) indicates padding may be specified

(P*) indicates that padding may be based on the number of lines affected

Name	Type	Pad?	Description
ae	str	(P)	End alternate character set
al	str	(P*)	Add new blank line
am	bool		Terminal has automatic margins
as	str	(P)	Start alternate character set
bc	str		Backspace if not ^H
bs	bool		Terminal can backspace with ^H
bt	str	(P)	Back tab
bw	bool		Backspace wraps from column 0 to last column
CC	str		Command character in prototype if terminal settable
cd	str	(P*)	Clear to end of display
ce	str	(P)	Clear to end of line
ch	str	(P)	Like cm but horizontal motion only, line stays same
cl	str	(P*)	Clear screen
cm	str	(P)	Cursor motion

co	num		Number of columns in a line
cr	str	(P*)	Carriage return, (default ^M)
cs	str	(P)	Change scrolling region (vt100), like cm
cv	str	(P)	Like ch but vertical only.
da	bool		Display may be retained above
dB	num		Number of millisec of bs delay needed
db	bool		Display may be retained below
dC	num		Number of millisec of cr delay needed
dc	str	(P*)	Delete character
dF	num		Number of millisec of ff delay needed
dl	str	(P*)	Delete line
dm	str		Delete mode (enter)
dN	num		Number of millisec of nl delay needed
do	str		Down one line
dT	num		Number of millisec of tab delay needed
ed	str		End delete mode
ei	str		End insert mode; give ":ei=:" if ic
eo	str		Can erase overstrikes with a blank
ff	str	(P*)	Hardcopy terminal page eject (default ^L)
hc	bool		Hardcopy terminal
hd	str		Half-line down (forward 1/2 linefeed)
ho	str		Home cursor (if no cm)
hu	str		Half-line up (reverse 1/2 linefeed)
hz	str		Hazeltine; can't print ~'s
ic	str	(P)	Insert character
if	str		Name of file containing is
im	bool		Insert mode (enter); give ":im=:" if ic
in	bool		Insert mode distinguishes nulls on display
ip	str	(P*)	Insert pad after character inserted
is	str		Terminal initialization string
k0-k9	str		Sent by "other" function keys 0-9
kb	str		Sent by backspace key
kd	str		Sent by terminal down arrow key
ke	str		Out of "keypad transmit" mode
kh	str		Sent by home key
kl	str		Sent by terminal left arrow key
kn	num		Number of "other" keys
ko	str		Termcap entries for other non-function keys
kr	str		Sent by terminal right arrow key
ks	str		Put terminal in "keypad transmit" mode
ku	str		Sent by terminal up arrow key
l0-l9	str		Labels on "other" function keys
li	num		Number of lines on screen or page
ll	str		Last line, first column (if no cm)
ma	str		Arrow key map, used by vi version 2 only
md	str		Enter bold mode
me	str		Turn off all attributes, normal mode
mh	str		Enter dim mode
mi	bool		Safe to move while in insert mode
ml	str		Memory lock on above cursor.
mr	str		Enter reverse mode
ms	bool		Safe to move while in standout and underline mode
mu	str		Memory unlock (turn off memory lock).

nc	bool		No correctly working carriage return (DM2500,H2000)
nd	str		Non-destructive space (cursor right)
nl	str	(P*)	Newline character (default \n)
ns	bool		Terminal is a CRT but doesn't scroll.
os	bool		Terminal overstrikes
pc	str		Pad character (rather than null)
pt	bool		Has hardware tabs (may need to be set with is)
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by so or se
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)
ta	str	(P)	Tab (other than ^I or with padding)
tc	str		Entry of similar terminal - must be last
te	str		String to end programs that use cm
ti	str		String to begin programs that use cm
uc	str		Underscore one char and move past it
ue	str		End underscore mode
ug	num		Number of blank chars left by us or ue
ul	bool		Terminal underlines even though it doesn't overstrike
up	str		Upline (cursor up)
us	str		Start underscore mode
vb	str		Visible bell (may not move cursor)
ve	str		Sequence to end open/visual mode
vs	str		Sequence to start open/visual mode
xb	bool		Beehive (f1=escape, f2=ctrl C)
xn	bool		A newline is ignored after a wrap (Concept)
xr	bool		Return acts like ce \r \n (Delta Data)
xs	bool		Standout not erased by writing over it (HP 264?)
xt	bool		Tabs are destructive, magic so char (Teleray 1061)

A Sample Entry

The following entry, which describes the Concept–100, is among the more complex entries in the *termcap* file as of this writing. This particular concept entry is outdated, and is used as an example only.

```
c1|c100|concept100:is=\EU\Ef\E7\E5\E8\El\ENH\EK\E\200\Eo&\200:\
	:al=3*\E^R:am:bs:cd=16*\E^C:ce=16\E^S:cl=2*^L:cm=\Ea%+ %+ :co#80:\
	:dc=16\E^A:dl=3*\E^B:ei=\E\200:eo:im=\E^P:in:ip=16*:li#24:mi:nd=\E=:\
	:se=\Ed\Ee:so=\ED\EE:ta=8\t:ul:up=\E;:vb=\Ek\EK:xn:
```

Entries may continue onto multiple lines by giving a \ as the last character of a line, and empty fields may be included for readability (here between the last field on a line and the first field on the next).

Types of Capabilities

Capabilities in *termcap* are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations. All capabilities have two letter codes.

Boolean capabilities are introduced simply by stating the two-character capability code in the field between ':' characters. For instance, the fact that the Concept has "automatic margins" (that is, an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**.

Numeric capabilities are followed by the character '#' and then the value. Thus **co** which indicates the number of columns the terminal has gives the value '80' for the Concept.

String valued capabilities, such as **ce** (clear to end of line sequence) are given by the two character code, an '=', and then a string ending at the next following ':'. A delay in milliseconds may appear after the '=' in such a capability, and padding characters are supplied by the editor after the remainder of the string is sent to provide this delay. The delay can be either a integer, for instance, '20', or an integer followed by an '*', that is, '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' to specify a delay per unit to tenths of milliseconds.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. A **\E** maps to an ESCAPE character, **^x** maps to a control-x for any appropriate x, and the sequences **\n \r \t \b \f** give a newline, return, tab, backspace and formfeed. Finally, characters may be given as three octal digits after a ****, and the characters **^** and **** may be given as **\^** and ****. If it is necessary to place a **:** in a capability it must be escaped in octal as **\072**. If it is necessary to place a null character in a string capability it must be encoded as **\200**. The routines which deal with *termcap* use C strings, and strip the high bits of the output very late so that a **\200** comes out as a **\000** would.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *termcap* and to build up a description gradually, using partial descriptions with *ex* to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the *termcap* file to describe it or bugs in *ex*. To easily test a new terminal description you can set the environment variable TERMCAP to a pathname of a file containing the description you are working on and the editor will look there rather than in */etc/termcap*. TERMCAP can also be set to the termcap entry itself to avoid reading the file when starting up the editor.

Basic capabilities

The number of columns on each line for the terminal is given by the **co** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **li** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, then this is given by the **cl** string capability. If the terminal can backspace, then it should have the **bs** capability, unless a backspace is accomplished by a character other than **^H** (ugh) in which case you should give this character as the **bc** string capability. If it overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability.

A very important point here is that the local cursor motions encoded in *termcap* are undefined at the left and top edges of a CRT terminal. The editor will never attempt to backspace around the left edge, nor will it attempt to go up locally off the top. The editor assumes that feeding off the bottom of the screen will cause the screen to scroll up, and the **am** capability tells whether the cursor sticks at the right edge of the screen. If the terminal has switch selectable automatic margins, the *termcap* file usually assumes that this is on, that is, **am**.

These capabilities suffice to describe hardcopy and "glass-tty" terminals. Thus the model 33 teletype is described as

```
t3|33|tty33:co#72:os
```

while the Lear Siegler ADM-3 is described as cl|adm3|3|lsi adm3:am:bs:cl=^Z:li#24:co#80

Cursor addressing

Cursor addressing in the terminal is described by a **cm** string capability, with *printf*(3S) like escapes **%x** in it. These substitute to encodings of the current line or column position, while other characters are passed through unchanged. If the **cm** string is thought of as being a function, then its arguments are the line and then the column to which motion is desired, and the **%** encodings have the following meanings:

%d	as in *printf*, 0 origin
%2	like %2d
%3	like %3d
%.	like %c
%+x	adds *x* to value, then %.
%>xy	if value > x adds y, no output.
%r	reverses order of line and column, no output
%i	increments line/column (for 1 origin)
%%	gives a single %
%n	exclusive or row and column with 0140 (DM2500)
%B	BCD (16*(x/10)) + (x%10), no output.
%D	Reverse coding (x-2*(x%16)), no output. (Delta Data).

Consider the HP2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cm** capability is "cm=6\E&%r%2c%2Y". The Microterm ACT-IV needs the current row and column sent preceded by a **^T**, with the row and column simply encoded in binary, "cm=^T%.%.". Terminals which use "%." need to be able to backspace the cursor (**bs** or **bc**), and to move the cursor up one line on the screen (**up** introduced below). This is necessary because it is not always safe to transmit **\t**, **\n** **^D** and **\r**, as the system may change or discard them.

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus "cm=\E=%+ %+ ".

Cursor motions

If the terminal can move the cursor one position to the right, leaving the character at the current position unchanged, then this sequence should be given as **nd** (non-destructive space). If it can move the cursor up a line on the screen in the same column, this should be given as **up**. If the terminal has no cursor addressing capability, but can home the cursor (to very upper left corner of screen) then this can be given as **ho**; similarly a fast way of getting to the lower left hand corner can be given as **ll**; this may involve going up with **up** from the home position, but the editor will never do this itself (unless **ll** does) because it makes no assumption about the effect of moving up from the home position.

Area clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **ce**. If the terminal can clear from the current position to the end of the display, then this should be given as **cd**. The editor only uses **cd** from the first column of a line.

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **al**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl**; this is done only from the first position on the line to be deleted. If the terminal can scroll the screen backwards, then this can be given as **sb**, but just **al** suffices. If the terminal can retain display memory above then the **da** capability should be given; if display memory can be retained below then **db** should be given. These let the editor understand that deleting a line on the screen may bring non-blank lines up from below or that scrolling back with **sb** may bring down non-blank lines.

Insert/delete character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using *termcap*. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can find out which kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type "abc   def" using local cursor motions (not spaces) between the "abc" and the "def". Then position the cursor before the "abc" and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the "abc" shifts over to the "def" which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for "insert null". If your terminal does something different and unusual then you may have to modify the editor to get it to use the insert mode your terminal defines. We have seen no terminals which have an insert mode not not falling into one of these two classes.

The editor can handle both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **im** the sequence to get into insert mode, or give it an empty value if your terminal uses a sequence to insert a blank position. Give as **ei** the sequence to leave insert mode (give this, with an empty value also if you gave **im** so). Now give as **ic** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ic**, terminals which send a sequence to open a screen position should give it here. (Insert mode is preferable to the sequence to open a position on the screen if your terminal has both.) If post insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (for example, if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mi** to speed up inserting in this case. Omitting **mi** will affect only speed. Some terminals (notably Datamedia's) must not have **mi** because of the way their insert mode works.

Finally, you can specify delete mode by giving **dm** and **ed** to enter and exit delete mode, and **dc** to delete a single character while in delete mode.

Highlighting, underlining, and visible bells

If your terminal has sequences to enter and exit standout mode these can be given as **so** and **se** respectively. If there are several flavors of standout mode (such as inverse video, blinking, or underlining – half bright is not usually an acceptable "standout" mode unless the terminal is in inverse video mode constantly) the preferred mode is inverse video by itself. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **ug** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **us** and **ue** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as **uc**. (If the underline code does not move the cursor to the right, give the code followed by a nondestructive space.)

Many terminals, such as the HP 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **vb**; it must not move the cursor. If the terminal should be placed in a different mode during open and visual modes of *ex*, this can be given as **vs** and **ve**, sent at the start and end of these modes respectively. These can be used to change, for example, from a underline to a block cursor and back.

If the terminal needs to be in a special mode when running a program that addresses the cursor, the codes to enter and exit this mode can be given as **ti** and **te**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly.

If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

ANSI terminals have modes for the character highlighting. Dim characters may be generated in dim mode, entered by **mh**; reverse video characters in reverse mode, entered by **mr**; bold characters in bold mode, entered by **md**; and normal mode characters restored by turning off all attributes with **me**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **ks** and **ke**. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kl, kr, ku, kd,** and **kh** respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be given as **k0, k1, ..., k9**. If these keys have labels other than the default f0 through f9, the labels can be given as **l0, l1, ..., l9**. If there are other keys that transmit the same code as the terminal expects for the corresponding function, such as clear screen, the *termcap* 2 letter codes can be given in the **ko** capability, for example, ":ko=cl,ll,sf,sb:", which says that the terminal has clear, home down, scroll down, and scroll up keys that transmit the same thing as the cl, ll, sf, and sb entries.

The **ma** entry is also used to indicate arrow keys on terminals which have single character arrow keys. It is obsolete but still in use in version 2 of vi, which must be run on some minicomputers due to memory limitations. This field is redundant with **kl, kr, ku, kd,** and **kh**. It consists of groups of two characters. In each group, the first character is what an arrow key sends, the second character is the corresponding vi command. These commands are **h** for **kl**, **j** for **kd**, **k** for **ku**, **l** for **kr**, and **H** for **kh**. For example, the mime would be **:ma=^KJ^Zk^Xl:** indicating arrow keys left (^H), down (^K), up (^Z), and right (^X). (There is no home key on the mime.)

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pc**.

If tabs on the terminal require padding, or if the terminal uses a character other than **^I** to tab, then this can be given as **ta**.

Hazeltine terminals, which don't allow '~' characters to be printed should indicate **hz**. Datamedia terminals, which echo carriage-return linefeed for carriage return and then ignore a following linefeed should indicate **nc**. Early Concept terminals, which ignore a linefeed immediately after an **am** wrap, should indicate **xn**. If an erase-eol is required to get rid of standout (instead of merely writing on top of it), **xs** should be given. Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt**. Other specific terminal problems may be corrected by adding more capabilities of the form **x***x*.

Other capabilities include **is**, an initialization string for the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to properly clear and then set the tabs on the terminal, if the terminal has settable tabs. If both are given, **is** will be printed before **if**. This is useful where **if** is */usr/lib/tabset/std* but **is** clears the tabs first.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **tc** can be given with the name of the similar terminal. This capability must be *last* and the combined length of the two entries must not exceed 1024. Since *termlib* routines search the entry from left to right, and since the tc capability is replaced by the corresponding entry, the capabilities given at the left override the ones in the similar terminal. A capability can be canceled with **xx@** where xx is the capability. For example, the entry

```
hn|2621nl:ks@:ke@:tc=2621:
```

defines a 2621nl that does not have the **ks** or **ke** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES

/etc/termcap file containing terminal descriptions

SEE ALSO

ex(1), curses(3X), termcap(3X), tset(1), vi(1), ul(1), more(1)

BUGS

Ex allows only 256 characters for string capabilities, and the routines in *termcap*(3X) do not check for overflow of this buffer. The total length of a single entry (excluding only escaped newlines) may not exceed 1024.

The **ma**, **vs**, and **ve** entries are specific to the *vi* program.

Not all programs support all entries. There are entries that are not supported by any program.

NAME

tp – DEC/mag tape formats

DESCRIPTION

Tp dumps files to and extracts files from DECtape and magtape. The formats of these tapes are the same except that magtapes have larger directories.

Block zero contains a copy of a stand-alone bootstrap program. See *reboot*(8).

Blocks 1 through 24 for DECtape (1 through 62 for magtape) contain a directory of the tape. There are 192 (resp. 496) entries in the directory; 8 entries per block; 64 bytes per entry. Each entry has the following format:

```
struct {
        char            pathname[32];
        unsigned short  mode;
        char            uid;
        char            gid;
        char            unused1;
        char            size[3];
        long            modtime;
        unsigned short  tapeaddr;
        char            unused2[16];
        unsigned short  checksum;
};
```

The path name entry is the path name of the file when put on the tape. If the pathname starts with a zero word, the entry is empty. It is at most 32 bytes long and ends in a null byte. Mode, uid, gid, size and time modified are the same as described under i-nodes (see file system *fs*(5)). The tape address is the tape block number of the start of the contents of the file. Every file starts on a block boundary. The file occupies (size+ 511)/512 blocks of continuous tape. The checksum entry has a value such that the sum of the 32 words of the directory entry is zero.

Blocks above 25 (resp. 63) are available for file storage.

A fake entry has a size of zero.

SEE ALSO

fs(5), tp(1)

BUGS

The *pathname, uid, gid,* and *size* fields are too small.

NAME

ttys – terminal initialization data

DESCRIPTION

The *ttys* file is read by the *init* program and specifies which terminal special files are to have a process created for them so that people can log in. There is one line in the *ttys* file per special file associated with a terminal.

The first character of a line in the *ttys* file is either '0' or '1'. If the first character on the line is a '0', the *init* program ignores that line. If the first character on the line is a '1', the *init* program creates a login process for that line.

The second character on each line is used as an argument to *getty*(8), which performs such tasks as baud-rate recognition, reading the login name, and calling *login*. For normal lines, the second character is '0'; other characters can be used, for example, with hard-wired terminals where speed recognition is unnecessary or which have special characteristics. The remainder of the line is the terminal's entry in the device directory, /dev.

Getty uses the second character in the *ttys* file to look up the characteristics of the terminal in the */etc/gettytab* file. Consult the *gettytab*(5) manual page for an explanation of the layout of */etc/gettytab*.

FILES

/etc/ttys

SEE ALSO

init(8), getty(8), login(1), gettytab(5)

NAME

ttytype – data base of terminal types by port

SYNOPSIS

/etc/ttytype

DESCRIPTION

Ttytype is a database containing, for each tty port on the system, the kind of terminal that is attached to it. There is one line per port, containing the terminal kind (as a name listed in termcap (5)), a space, and the name of the tty, minus /dev/.

This information is read by *tset*(1) and by *login*(1) to initialize the TERM variable at login time.

SEE ALSO

tset(1), login(1)

BUGS

Some lines are merely known as "dialup" or "plugboard".

NAME

uuencode – format of an encoded uuencode file

DESCRIPTION

Files output by *uuencode(1C)* consist of a header line, followed by a number of body lines, and a trailer line. *Uudecode* will ignore any lines preceding the header or following the trailer. Lines preceding a header must not, of course, look like a header.

The header line is distinguished by having the first 6 characters "begin ". The word *begin* is followed by a mode (in octal), and a string which names the remote file. Spaces separate the three items in the header line.

The body consists of a number of lines, each at most 62 characters long (including the trailing newline). These consist of a character count, followed by encoded characters, followed by a newline. The character count is a single printing character, and represents an integer, the number of bytes the rest of the line represents. Such integers are always in the range from 0 to 63 and can be determined by subtracting the character space (octal 40) from the character.

Groups of 3 bytes are stored in 4 characters, 6 bits per character. All are offset by a space to make the characters printing. The last line may be shorter than the normal 45 bytes. If the size is not a multiple of 3, this fact can be determined by the value of the count on the last line. Extra garbage will be included to make the character count a multiple of 4. The body is terminated by a line with a count of zero. This line consists of one ASCII space.

The trailer line consists of "end" on a line by itself.

SEE ALSO

uuencode(1C), uudecode(1C), uusend(1C), uucp(1C), mail(1)

NAME

vfont – font formats

SYNOPSIS

#include <vfont.h>

DESCRIPTION

The fonts used by the window system and printer/plotters have the following format. Each font is in a file, which contains a header, an array of character description structures, and an array of bytes containing the bit maps for the characters. The header has the following format:

```
struct header {
        short           magic;          /* Magic number VFONT_MAGIC */
        unsigned short  size;           /* Total # bytes of bitmaps */
        short           maxx;           /* Maximum horizontal glyph size */
        short           maxy;           /* Maximum vertical glyph size */
        short           xtend;          /* (unused) */
};
#define VFONT_MAGIC                     0436
```

Maxx and *maxy* are intended to be the maximum horizontal and vertical size of any glyph in the font, in raster lines. (A glyph is just a printed representation of a character, in a particular size and font.) The *size* is the total size of the bit maps for the characters in bytes. The *xtend* field is not currently used.

After the header is an array of NUM_DISPATCH structures, one for each of the possible characters in the font. Each element of the array has the form:

```
struct dispatch {
        unsigned short  addr;                   /* &(glyph) - &(start of bitmaps) */
        short           nbytes;                 /* # bytes of glyphs (0 if no glyph) */
        char            up, down, left, right;  /* Widths from baseline point */
        short           width;                  /* Logical width, used by troff */
};
#define NUM_DISPATCH                            256
```

The *nbytes* field is nonzero for characters which actually exist. For such characters, the *addr* field is an offset into the bit maps to where the character's bit map begins. The *up, down, left,* and *right* fields are offsets from the base point of the glyph to the edges of the rectangle which the bit map represents. (The imaginary "base point" is a point which is vertically on the "base line" of the glyph (the bottom line of a glyph which doesn't have a descender) and horizontally near the left edge of the glyph; often 3 or so pixels past the left edge.) The bit map contains *up+ down* rows of data for the character, each of which has *left+ right* columns (bits). Each row is rounded up to a number of bytes. The *width* field represents the logical width of the glyph in bits, and shows the horizontal displacement to the base point of the next glyph.

FILES

/usr/lib/vfont/*
/usr/suntool/fixedwidthfonts/*

SEE ALSO

troff(1), pti(1), vfontinfo(1), vswap(1)

BUGS

A machine-independent font format should be defined. The **shorts** in the above structures contain different bit patterns depending whether the font file is for use on a Vax or a Sun. The *vswap* program must be used to convert one to the other.

Interprocess Communication Primer

Table of Contents

Interprocess Communication Primer

This document provides an introduction to the interprocess communication facilities included in the Sun Workstation version of the UNIX† operating system.

It discusses the overall model for interprocess communication and introduces the interprocess communication primitives which have been added to the system. The majority of the document considers the use of these primitives in developing applications. The reader is expected to be familiar with the C programming language as all examples are written in C.

† UNIX is a trademark of Bell Laboratories.

1. Introduction

One of the most important features added in the Berkeley 4.2 release of the UNIX operating system is substantial new interprocess communication facilities. These facilities are the result of more than two years of discussion and research. The facilities provided in this release incorporate many of the ideas from current research, while trying to maintain the UNIX philosophy of simplicity and conciseness. We hope that these interprocess communication facilities will establish a standard. From the response to the design, it appears that it is being adopted on many systems.

UNIX has previously been very weak in the area of interprocess communication. Until recently, the only standard mechanism which allowed two processes to communicate were pipes (the mpx files which were part of Version 7 were experimental). Unfortunately, pipes are restrictive in that the two communicating processes must be related through a common ancestor. Further, the semantics of pipes makes them almost impossible to maintain in a distributed environment.

Earlier attempts at extending the ipc facilities of UNIX have met with mixed reaction. The majority of the problems have been related to the fact these facilities have been tied to the UNIX file system; either through naming, or implementation. Consequently, the ipc facilities provided in this release have been designed as a totally independent subsystem, and allow processes to rendezvous in many ways. Processes may rendezvous through a UNIX file system-like name space (a space where all names are path names) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, the communication facilities have been extended to included more than the simple byte stream provided by a pipe-like entity. These extensions have resulted in a completely new part of the system which users will need time to familiarize themselves with. It is likely that as more use is made of these facilities they will be refined; only time will tell.

The remainder of this document is organized in four sections. Section 2 introduces the new system calls and the basic model of communication. Section 3 describes some of the supporting library routines users may find useful in constructing distributed applications. Section 4 is concerned with the client/server model used in developing applications and includes examples of the two major types of servers. Section 5 delves into advanced topics which sophisticated users are likely to encounter when using the ipc facilities.

2. Basics

The basic building block for communication is the *socket*. A socket is an endpoint of communication to which a name may be *bound*. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the UNIX communication domain sockets are named with UNIX path names; e.g. a socket may be named "/dev/foo". Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The ipc supports two separate communication domains: the UNIX domain, and the Internet domain is used by processes which communicate using the the DARPA standard communication protocols. The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket "operating" in the UNIX domain sees a subset of the possible error conditions which are possible when operating in the Internet domain.

2.1. Socket Types

Sockets are typed according to the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Three types of sockets currently are available to a user. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes*.

A *datagram* socket supports bidirectional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol.

Two potential socket types which have interesting properties are the *sequenced packet* socket and the *reliably delivered message* socket. A sequenced packet socket is identical to a stream socket with the exception that record boundaries are preserved. This interface is very similar to that provided by the Xerox NS Sequenced Packet protocol. The reliably delivered message socket has similar properties to a datagram socket, but with reliable delivery. While these two socket types have been loosely defined, they are not currently implemented. So, in this

* In the UNIX domain, in fact, the semantics are identical and, as one might expect, pipes have been implemented internally as simply a pair of connected stream sockets.

document, we will concern ourselves only with the three supported socket types.

2.2. Socket Creation

To create a socket the *socket* system call is used:

```
s = socket(domain, type, protocol);
```

This call requests that the system create a socket in the specified *domain* and of the specified *type*. A particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type. The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets. The domain is specified as one of the manifest constants defined in the file *<sys/socket.h>*. For the UNIX domain the constant is AF_UNIX*; for the Internet domain AF_INET. The socket types are also defined in this file and one of SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW must be specified. To create a stream socket in the Internet domain the following call might be used:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for on-machine use a sample call might be:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

To obtain a particular protocol one selects the protocol number, as defined within the communication domain. For the Internet domain the available protocols are defined in *<netinet/in.h>* or, better yet, one may use one of the library routines discussed in section 3, such as *getprotobyname*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("tcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

* The manifest constants are named AF_whatever as they indicate the "address format" to use in interpreting names.

2.3. Binding Names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. The *bind* call is used to assign a name to a socket:

```
bind(s, name, namelen);
```

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties which comprise the "domain"). In the UNIX domain names are path names while in the Internet domain names contain an Internet address and port number. If one wanted to bind the name "/dev/foo" to a UNIX domain socket, the following would be used:

```
#include <sys/un.h>
struct sockaddr_un sun;
sun.sun_family = AF_UNIX;
strcpy(sun.sun_path, "/dev/foo");
bind(s, &sun, strlen("/dev/foo")+ 2);
```

In binding an Internet address things become more complicated. The actual call is simple,

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
bind(s, &sin, sizeof (sin));
```

but the selection of what to place in the address *sin* requires some discussion. We will come back to the problem of formulating Internet addresses in section 3 when the library routines used in name resolution are discussed.

2.4. Connection Establishment

With a bound socket it is possible to rendezvous with an unrelated process. This operation is usually asymmetric with one process a "client" and the other a "server". The client requests services from the server by initiating a "connection" to the server's socket. The server, when willing to offer its advertised services, passively "listens" on its socket. On the client side the *connect* call is used to initiate a connection. Using the UNIX domain, this might appear as,

```
struct sockaddr_un server;
connect(s, &server, strlen(server.sun_path)+ 2);
```

while in the Internet domain,

```
struct sockaddr_in server;
connect(s, &server, sizeof (server));
```

If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket; c.f. section 5.4[1]. An error is returned when

[1] You must do a *getsockname*(2) call to retrieve the binding.

the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin.

Many errors can be returned when a connection attempt fails. The most common are:

ETIMEDOUT
: After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED
: The host refused service for some reason. When connecting to a host running the 0.9 release version of UNIX this is usually due to a server process not being present at the requested name.

ENETDOWN or EHOSTDOWN
: These operational errors are returned based on status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH
: These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down. In these cases the system is conservative and indicates the entire network is unreachable.

For the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the *listen* call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages which comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may *accept* a connection:

```
fromlen = sizeof (from);
snew = accept(s, &from, &fromlen);
```

A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be zero.

Accept normally blocks. That is, the call to accept will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or not block on the accept call there are alternatives; they will be considered in section 5.

2.5. Data Transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect, in this case, then the normal *read* and *write* system calls are useable,

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to *read* and *write*, the new calls *send* and *recv* may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While *send* and *recv* are virtually identical to *read* and *write*, the extra *flags* argument is important. The flags may be specified as a non-zero value if one or more of the following is required:

MSG_OOB	send/receive out of band data
MSG_PEEK	look at data without reading
MSG_DONTROUTE	send data without routing packets

Out of band data is a notion specific to stream sockets, and one which we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When MSG_PREVIEW is specified with a *recv* call, any data present is returned to the user, but treated as still "unread". That is, the next *read* or *recv* call applied to the socket will return the data previously previewed.

2.6. Discarding Sockets

Once a socket is no longer of interest, it may be discarded by applying a *close* to the descriptor,

```
close(s);
```

If data is associated with a socket which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it may perform a *shutdown* on the socket prior to closing it. This call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received. Applying shutdown to a socket causes any data queued to be immediately discarded.

2.7. Connectionless Sockets

To this point we have been concerned mostly with sockets which follow a connection oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before, and each should have a name bound to it in order that the recipient of a message may identify the sender. To send data, the *sendto* primitive is used,

```
sendto(s, buf, buflen, flags, &to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the intended recipient of the message. When using an unreliable datagram interface, it is unlikely any errors will be reported to the sender. Where information is present locally to recognize a message which may never be delivered (for instance when a network is unreachable), the call will return -1 and the global value *errno* will contain an error number.

To receive messages on an unconnected datagram socket, the *recvfrom* primitive is provided:

```
recvfrom(s, buf, buflen, flags, &from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer.

In addition to the two calls mentioned above, datagram sockets may also use the *connect* call to associate a socket with a specific address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket (i.e. no multicasting). Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket where a connect request initiates establishment of an end to end connection). Other of the less important details of datagram sockets are described in section 5.

2.8. Input/Output Multiplexing

One last facility often used in developing applications is the ability to multiplex i/o requests among multiple sockets and/or files. This is done using the *select* call:

```
select(nfds, &readfds, &writefds, &execptfds, &timeout);
```

Select takes as arguments three bit masks, one for the set of file descriptors for which the caller wishes to be able to read data on, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending. Bit masks are created by or-ing bits of the form "1 << fd". That is, a descriptor *fd* is selected if a 1 is present in the *fd*'th bit of the mask. The parameter *nfds* specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) specified in a mask.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If *timeout* is set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely*. *Select*

[1] * To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

normally returns the number of file descriptors selected. If the *select* call returns due to the timeout expiring, then a value of -1 is returned along with the error number EINTR.

Select provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in section 5.

3. Network Library Routines

The discussion in section 2 indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task a number of routines have been added to the standard C run-time library. In this section we will consider the new routines provided to manipulate network addresses. While the Sun system release networking facilities support only the DARPA standard Internet protocols, these routines have been designed with flexibility in mind. As more communication protocols become available, we hope the same user interface will be maintained in accessing network-related address data bases. The only difference should be the values returned to the user. Since these values are normally supplied the system, users should not need to be directly aware of the communication protocol and/or naming conventions in use.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; e.g. "the *login server* on host monet". This name, and the name of the peer host, must then be translated into network *addresses* which are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings is likely to vary between network architectures. For instance, it is desirable for a network to not require hosts be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for: mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file <*netdb.h*> must be included when using any of these routines.

3.1. Host Names

A host name to address mapping is represented by the *hostent* structure:

```
struct  hostent {
        char      *h_name;        /* official name of host */
        char      **h_aliases;    /* alias list */
        int       h_addrtype;     /* host address type */
        int       h_length;       /* length of address */
        char      *h_addr;        /* address */
};
```

Note that the *h_addr* field in the structure definition is defined as a pointer to **char**. In the case of Internet addresses (the only case implemeted to date) you should cast this to a (**struct** in_addr *) when using the item.

The official name of the host and its public aliases are returned, along with a variable length address and address type. The routine *gethostbyname*(3N) takes a host name and returns a *hostent* structure, while the routine *gethostbyaddr*(3N) maps host addresses into a *hostent* structure. It is possible for a host to have many addresses, all having the same name. *Gethostybyname* returns the first matching entry in the data base file */etc/hosts*; if this is unsuitable, the lower level routine *gethostent*(3N) may be used. For example, to obtain a *hostent* structure for a host on a particular network the following routine might be used (for simplicity, only Internet addresses are considered):

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
struct hostent *
gethostbynameandnet(name, net)
        char *name;
        int net;
{
        register struct hostent *hp;
        register char **cp;

        sethostent(0);
        while ((hp = gethostent()) != NULL) {
                if (hp->h_addrtype != AF_INET)
                        continue;
                if (strcmp(name, hp->h_name)) {
                        for (cp = hp->h_aliases; cp && *cp != NULL; cp++)
                                if (strcmp(name, *cp) == 0)
                                goto found;
                        continue;
                }
        found:
                if (in_netof(*(struct in_addr *)hp->h_addr)) == net)
                        break;
        }
        endhostent(0);
        return (hp);
}
```

(*in_netof*(3N) is a standard routine which returns the network portion of an Internet address.)

3.2. Network Names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct  netent {
        char      *n_name;        /* official name of net */
        char      **n_aliases;    /* alias list */
        int       n_addrtype;     /* net address type */
        int       n_net;          /* network # */
};
```

The routines *getnetbyname*(3N), *getnetbynumber*(3N), and *getnetent*(3N) are the network counterparts to the host routines described above.

3.3. Protocol Names

For protocols the *protoent* structure defines the protocol-name mapping used with the routines *getprotobyname*(3N), *getprotobynumber*(3N), and *getprotoent*(3N):

```
struct  protoent {
        char      *p_name;        /* official protocol name */
        char      **p_aliases;    /* alias list */
        int       p_proto;        /* protocol # */
};
```

3.4. Service Names

Information regarding services is a bit more complicated. A service is expected to reside at a specific "port" and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports or support multiple protocols. If either of these occurs, the higher level library routines will have to be bypassed in favor of homegrown routines similar in spirit to the "gethostbynameandnet" routine described above. A service mapping is described by the *servent* structure,

```
struct  servent {
        char      *s_name;        /* official service name */
        char      **s_aliases;    /* alias list */
        int       s_port;         /* port # */
        char      *s_proto;       /* protocol to use */
};
```

The routine *getservbyname*(3N) maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

```
sp = getservbyname("telnet", (char *)0);
```

returns the service specification for a telnet server using any protocol, while the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routines *getservbyport*(3N) and *getservent*(3N) are also provided. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; an optional protocol name may be specified to qualify lookups.

3.5. Miscellaneous

With the support routines described above, an application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in Figure 1. (This example will be considered in more detail in section 4.)

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be worthwhile. Perhaps when the system is adapted to different network architectures the utilities will be reorganized more cleanly.

Aside from the address-related data base routines, there are several other routines available in the run-time library which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. Table 1 summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On a VAX, or machine with similar architecture, this is usually reversed. Consequently, programs are sometimes required to byte swap quantities. The library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
 ...
main(argc, argv)
        char *argv[];
{
        struct sockaddr_in sin;
        struct servent *sp;
        struct hostent *hp;
        int s;
        ...
        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service\n");
                exit(1);
        }
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }
        bzero((char *)&sin, sizeof (sin));
        bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
        sin.sin_family = hp->h_addrtype;
        sin.sin_port = sp->s_port;
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
        ...
        if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }
        ...
}
```

Figure 1. Remote login client code.

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines other than the VAX these routines are defined as null macros.

Call	Synopsis
bcmp(s1, s2, n)	compare byte-strings; 0 if same, not 0 otherwise
bcopy(s1, s2, n)	copy n bytes from s1 to s2
bzero(base, n)	zero-fill n bytes starting at base
htonl(val)	convert 32-bit quantity from host to network byte order
htons(val)	convert 16-bit quantity from host to network byte order
ntohl(val)	convert 32-bit quantity from network to host byte order
ntohs(val)	convert 16-bit quantity from network to host byte order

Table 1. C run-time routines.

4. Client/Server Model

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server which has been examined in section 2. In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

Client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a "client process" and a "server process". We will first consider the properties of server processes, then client processes.

A server process normally listens at a well know address for service requests. Alternative schemes which use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. The Xerox Courier protocol uses the latter scheme. When using Courier, a Courier client process contacts a Courier server at the remote host and identifies the service it requires. The Courier server process then creates the appropriate server process based on a data base and "splices" the client and server together, voiding its part in the transaction. This scheme is attractive in that the Courier server process may provide a single contact point for all services, as well as carrying out the initial steps in authentication. However, while this is an attractive possibility for standardizing access to services, it does introduce a certain amount of overhead due to the intermediate process involved. Implementations which provide this type of service within the system can minimize the cost of client server rendezvous.

4.1. Servers

In this release, most servers are accessed at well known Internet addresses or UNIX domain names. When a server is started at boot time it advertises it services by listening at a well know location. For example, the remote login server's main loop is of the form shown in Figure 2.

```
main(argc, argv)
	int argc;
	char **argv;
{
	int f;
	struct sockaddr_in from;
	struct servent *sp;

	sp = getservbyname("login", "tcp");
	if (sp == NULL) {
		fprintf(stderr, "rlogind: tcp/login: unknown service\n");
		exit(1);
	}
	...
#ifndef DEBUG
	<<disassociate server from controlling terminal>>
#endif
	...
	sin.sin_port = sp->s_port;
	...
	f = socket(AF_INET, SOCK_STREAM, 0);
	...
	if (bind(f, (caddr_t)&sin, sizeof (sin)) < 0) {
		...
	}
	...
	listen(f, 5);
	for (;;) {
		int g, len = sizeof (from);

		g = accept(f, &from, &len);
		if (g < 0) {
			if (errno != EINTR)
				perror("rlogind: accept");
			continue;
		}
		if (fork() == 0) {
			close(f);
			doit(g, &from);
		}
		close(g);
	}
}
```

Figure 2. Remote login server.

The first step taken by the server is look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
```

```
                fprintf(stderr, "rlogind: tcp/login: unknown service\n");
                exit(1);
        }
```

This definition is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

Step two is to disassociate the server from the controlling terminal of its invoker. This is important as the server will likely not want to receive signals delivered to the process group of the controlling terminal.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its expected location. The main body of the loop is fairly simple:

```
    for (;;) {
            int g, len = sizeof (from);

            g = accept(f, &from, &len);
            if (g < 0) {
                    if (errno != EINTR)
                            perror("rlogind: accept");
                    continue;
            }
            if (fork() == 0) {
                    close(f);
                    doit(g, &from);
            }
            close(g);
    }
```

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in section 5). Therefore, the return value from *accept* is checked to insure a connection has actually been established. With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queueing connection requests is closed in the child, while the socket created as a result of the accept is closed in the parent. The address of the client is also handed the *doit* routine because it requires it in authenticating clients.

4.2. Clients

The client side of the remote login service was shown earlier in Figure 1. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client remote login process. As in the server process the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service\n");
        exit(1);
}
```

Next the destination host is looked up with a *gethostbyname* call:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login process resides:

```
bzero((char *)&sin, sizeof (sin));
bcopy(hp->h_addr, (char *)sin.sin_addr, hp->h_length);
sin.sin_family = hp->h_addrtype;
sin.sin_port = sp->s_port;
```

A socket is created, and a connection initiated.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
        perror("rlogin: socket");
        exit(3);
}
 ...
if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
        perror("rlogin: connect");
        exit(4);
}
```

The details of the remote login protocol will not be considered here.

4.3. Connectionless Servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the "rwho" service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the rwho server may find out the current status of a machine with the *ruptime*(1) program. The output generated is illustrated in Figure 3.

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

arpa	up	9:45,	5 users, load	1.15,	1.39,	1.31
cad	up	2+ 12:04,	8 users, load	4.67,	5.13,	4.59
calder	up	10:10,	0 users, load	0.27,	0.15,	0.14
dali	up	2+ 06:28,	9 users, load	1.04,	1.20,	1.65
degas	up	25+ 09:48,	0 users, load	1.49,	1.43,	1.41
ear	up	5+ 00:05,	0 users, load	1.51,	1.54,	1.56
ernie	down	0:24				
esvax	down	17:04				
ingres	down	0:26				
kim	up	3+ 09:16,	8 users, load	2.03,	2.46,	3.11
matisse	up	3+ 06:18,	0 users, load	0.03,	0.03,	0.05
medea	up	3+ 09:39,	2 users, load	0.35,	0.37,	0.50
merlin	down	19+ 15:37				
miro	up	1+ 07:20,	7 users, load	4.59,	3.28,	2.12
monet	up	1+ 00:43,	2 users, load	0.22,	0.09,	0.07
oz	down	16:09				
statvax	up	2+ 15:57,	3 users, load	1.52,	1.81,	1.86
ucbvax	up	9:34,	2 users, load	6.08,	5.16,	3.28

Figure 3. ruptime output.

The rwho server, in a simplified form, is pictured in Figure 4. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to insure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet does, however, indicates some problems with the current protocol.

Status information is broadcast on the local network. For networks which do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status received). This, unfortunately, requires some bootstrapping information, as a server started up on a quiet network will have no known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplying it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information*.

[1] * One must, however, be concerned about "loops". That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, ex-

```
main()
{
	...
	sp = getservbyname("who", "udp");
	net = getnetbyname("localnet");
	sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
	sin.sin_port = sp->s_port;
	...
	s = socket(AF_INET, SOCK_DGRAM, 0);
	...
	bind(s, &sin, sizeof (sin));
	...
	sigset(SIGALRM, onalrm);
	onalrm();
	for (;;) {
		struct whod wd;
		int cc, whod, len = sizeof (from);

		cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0, &from, &len);
		if (cc <= 0) {
			if (cc < 0 && errno != EINTR)
				perror("rwhod: recv");
			continue;
		}
		if (from.sin_port != sp->s_port) {
			fprintf(stderr, "rwhod: %d: bad from port\n",
				ntohs(from.sin_port));
			continue;
		}
		...
		if (!verify(wd.wd_hostname)) {
			fprintf(stderr, "rwhod: malformed host name from %x\n",
				ntohl(from.sin_addr.s_addr));
			continue;
		}
		(void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
		whod = open(path, O_FWRONLY|O_FCREATE|O_FTRUNCATE, 0666);
		...
		(void) time(&wd.wd_recvtime);
		(void) write(whod, (char *)&wd, cc);
		(void) close(whod);
	}
}
```

Figure 4. **rwho server.**

change of information.

The second problem with the current scheme is that the rwho process services only a single local network, and this network is found by reading a file. It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. The Sun system attempts to isolate host-specific information from applications by providing system calls which return the necessary information†. The rwho server performs a lookup in a file to find its local network. A better, though still unsatisfactory, scheme used by the routing process is to interrogate the system data structures to locate those directly connected networks. A mechanism to acquire this information from the system would be a useful addition.

[1] † An example of such a system call is the *gethostname*(2) call which returns the host's "official" name.

5. Advanced Topics

A number of facilities have yet to be discussed. For most users of the ipc the mechanisms already described will suffice in constructing distributed applications. However, others will find need to utilize some of the features which we consider in this section.

5.1. Out of Band Data

The stream socket abstraction includes the notion of "out of band" data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data along with the SIGURG signal. In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals from between client and server processes. When a signal is expected to flush any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

The stream abstraction defines that the out of band data facilities must support the reliable delivery of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (that is, the urgent data is delivered in sequence with the normal data) the system extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data.

To send an out of band message the MSG_OOB flag is supplied to a *send* or *sendto* calls, while to receive out of band data MSG_OOB should be indicated when performing a *recvfrom* or *recv* call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK ioctl is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 5.

5.2. Signals and Process Groups

Due to the existence of the SIGURG and SIGIO signals each socket has an associated process group (just as is done for terminals). This process group is initialized to the process group of its creator, but may be redefined at a later time with the SIOCSPGRP ioctl:

```
oob()
{
	int out = 1+1;
	char waste[BUFSIZ], mark;

	signal(SIGURG, oob);
	/* flush local terminal input and output */
	ioctl(1, TIOCFLUSH, (char *)&out);
	for (;;) {
		if (ioctl(rem, SIOCATMARK, &mark) < 0) {
			perror("ioctl");
			break;
		}
		if (mark)
			break;
		(void) read(rem, waste, sizeof (waste));
	}
	recv(rem, &mark, 1, MSG_OOB);
	...
}
```

Figure 5. Flushing terminal i/o on receipt of out of band data.

```
ioctl(s, SIOCSPGRP, &pgrp);
```

A similar ioctl, SIOCGPGRP, is available for determining the current process group of a socket.

5.3. Pseudo Terminals

Many programs will not function properly without a terminal for standard input and output. Since a socket is not a terminal, it is often necessary to have a process communicating over the network do so through a *pseudo terminal*. A pseudo terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo terminal is supplied as input to a process reading from the master side. Data written on the master side is given the slave as input. In this way, the process manipulating the master side of the pseudo terminal has control over the information read and written on the slave side. The remote login server uses pseudo terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends an interrupt or quit signal to a process executing on a remote machine, the client login program traps the signal, sends an out of band message to the server process who then uses the signal number, sent as the data value in the out of band message, to perform a *killpg*(2) on the appropriate process group.

5.4. Internet Address Binding

Binding addresses to sockets in the Internet domain can be fairly complex. Communicating processes are bound by an *association*. An association is composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each Internet protocol. Associations are always unique. That is, there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples.

The bind system call allows a process to specify half of an association, <local address, local port>, while the connect and accept primitives are used to complete a socket's association. Since the association is created in two steps the association uniqueness requirement indicated above could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding the notion of a "wildcard" address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in <netinet/in.h>), the system interprets the address as "any valid address". For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and addressed to any of the possible addresses assigned a host. For example, if a host is on networks 46 and 10 and a socket is bound as above, then an accept call is performed, the process will be able to accept connection requests which arrive either from network 46 or network 10.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. For example:

```
sin.sin_addr.s_addr = MYADDRESS;
sin.sin_port = 0;
bind(s, (char *)&sin, sizeof (sin));
```

The system selects the port number based on two criteria. The first is that ports numbered 0 through IPPORT_RESERVED−1 are reserved for privileged users (that is, the super user). The second is that the port number is not currently bound to some other socket. In order to find a free port number in the privileged range the following code is used by the remote shell server:

```
struct sockaddr_in sin;
 ...
lport = IPPORT_RESERVED - 1;
sin.sin_addr.s_addr = INADDR_ANY;
 ...
for (;;) {
      sin.sin_port = htons((u_short)lport);
      if (bind(s, (caddr_t)&sin, sizeof (sin)) >= 0)
            break;
      if (errno != EADDRINUSE && errno != EADDRNOTAVAIL) {
            perror("socket");
            break;
      }
      lport--;
      if (lport == IPPORT_RESERVED/2) {
            fprintf(stderr, "socket: All ports in use\n");
            break;
      }
}
```

The restriction on allocating ports was done to allow processes executing in a "secure" environment to perform authentication based on the originating address and port number.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. This is due to associations being created in a two step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket were around. To override the default port selection algorithm then an option call must be performed prior to address binding:

```
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)0, 0);
bind(s, (char *)&sin, sizeof (sin));
```

With the above call, local addresses may be bound which are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port (if an association already exists, the error EADDRINUSE is returned).

Local address binding by the system is currently done somewhat haphazardly when a host is on multiple networks. Logically, one would expect the system to bind the local address associated with the network through which a peer was communicating. For instance, if the local host is connected to networks 46 and 10 and the foreign host is on network 32, and traffic from network 32 were arriving via network 10, the local address to be bound would be the host's address on network 10, not network 46. This unfortunately, is not always the case. For reasons too complicated to discuss here, the local address bound may be appear to be chosen at random. This property of local address binding will normally be invisible to users unless the foreign host does not understand how to reach the address selected*.

[1] * For example, if network 46 were unknown to the host on network 32, and the local address were bound to that located on network 46, then even though a route between the two hosts existed through network 10, a connection would fail.

5.5. Broadcasting and Datagram Sockets

By using a datagram socket it is possible to send broadcast packets on many networks supported by the system (the network itself must support the notion of broadcasting; the system provides no broadcast simulation in software). Broadcast messages can place a high load on a network since they force every host on the network to service them.

To send a broadcast message, an Internet datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Then the message should be addressed as:

```
dst.sin_family = AF_INET;
inet_makeaddr(net, INADDR_ANY);
dst.sin_port = DESTPORT;
```

and, finally, a sendto call may be used:

```
sendto(s, buf, buflen, 0, &dst, sizeof (dst));
```

Received broadcast messages contain the senders address and port (datagram sockets are anchored before a message is allowed to go out).

There are a couple of minor problems in the above example. One is created because INADDR_ANY has two meanings:

1. Fill in my own address, and,
2. Broadcast.

Unfortunately, broadcast must at some time in the future be changed to -1 instead of 0, so that broadcast will no longer be The second problem is how do you get your net number? You could use the SIOCGICONF ioctl call, or you could get your own address and do a *inet_netof* on that. INADDR_ANY.

5.6. Signals

Two new signals have been added to the system which may be used in conjunction with the interprocess communication facilities. The SIGURG signal is associated with the existence of an "urgent condition". The SIGIO signal is used with "interrupt driven i/o" (not presently implemented). SIGURG is currently supplied a process when out of band data is present at a socket. If multiple sockets have out of band data awaiting delivery, a select call may be used to determine those sockets with such data.

An old signal which is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any children processes have changed state. Normally servers use the signal to "reap" child processes after exiting. For example, the remote login server loop shown in Figure 2 may be augmented as follows:

```
int reaper();
 ...
signal(SIGCHLD, reaper);
listen(f, 10);
for (;;) {
	int g, len = sizeof (from);

	g = accept(f, &from, &len, 0);
	if (g < 0) {
		if (errno != EINTR)
			perror("rlogind: accept");
		continue;
	}
	...
}
 ...
#include <wait.h>
reaper()
{
	union wait status;

	while (wait3(&status, WNOHANG, 0) > 0)
		;
}
```

If the parent server process fails to reap its children, a large number of "zombie" processes may be created.