Dansk Data Elektronik A/S

SUPERMAX

System V, Release 3.1

RISC SWD Reference Manual

Section 2 and 3

Version 4.1

Version 4.1, published March 1993

Printed in Denmark

Stock no.: 94306411

NOTICE

The information in this document is subject to change without notice. AT&T or Dansk Data Elektronik A/S, Denmark assumes no responsibility for any errors that may appear in this document.

Permuted Index

This is a permuted index of all the articles found in the *Supermax System V, SWD RISC Reference Manual, Version 4.1.*

The "Permuted Index" is a list of keywords, given in the second of three columns, together with the context in which each keyword is found.

Keywords are either topical keywords or the names of manual entries. Entries are identified with their section numbers shown in parentheses. This is important because there is considerable duplication of names among the sections, arising principally from commands and functions that exist only to exercise a particular system call. The right column lists the name of the manual page on which each keyword may be found. The left column contains useful information about the keyword.

Column 1)	A possibly empty 'head' field.
Column 2)	A 'key' field, followed by a number of periods.
Column 3)	A 'reference' field.

The index is sorted alphabetically by the key field.

Most lines in the index are taken directly from the 'NAME' section of each article. Each word of that short description of the article is used as a key in the key field.

The head field contains the part of the description preceding the key.

The reference field tells the reader where to find the article.

As an example consider the article about the *ls* in Section 1 of the Reference Manuals. The purpose of *ls* is to 'list contents of directory'. Therefore *ls* may be found in the permuted index in four places, namely under *ls*, under *list*, under *contents*, and under *directory*, thus:

The most common words, such as 'a', 'the', 'of', etc., are not used as keys.

Permuted Index

NAME

intro − introduction to libraries, functions, system calls and error numbers

SYNOPSIS

#include <errno.h>
#include <smoserr.h>

GENERAL DESCRIPTION

Sections 2 and 3 of this reference manual describes the C-language routines available in the *Supermax Operating System* System V. The routines may be divided into the following categories:

1) **System calls**. These routines are the low-level routines that call on the operating system to perform certain tasks. An example is the routine *open*, that opens a file.

2) **Subroutines**. Subroutines are auxiliary routines that make programming easier. They may or may not use system calls to perform their task.

DESCRIPTION

This section describes functions found in various libraries. Certain major collections are identified by a letter after the section number:

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library *libc*, which is automatically loaded by the C compiler, *cc*(1). (For this reason the (3C) and (3S) sections together comprise one section of this manual.) The link editor *ld*(1) searches this library under the **−lc** option. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.

(3S) These functions constitute the "standard I/O package" [see *stdio*(3S)]. These functions are in the library *libc*, already mentioned. Declarations for these functions may be obtained from the **#include** file **<stdio.h>**.

(3M) These functions constitute the Math Library, *libm*. They are not automatically loaded by the C compiler, *cc*(1); however, the link editor searches this library under the **−lm** option. Declarations for these functions may be obtained from the **#include** file **<math.h>**. Several generally useful mathematical constants are also defined there [see *math*(5)].

(3X) Various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

DEFINITIONS

A *character* is any bit pattern able to fit into a byte on the machine. The *null character* is a character with value 0, represented in the C language as '\0'. A *character array* is a sequence of characters. A *null-terminated character array* is a sequence of characters, the last of which is the *null character*. A *string* is a designation for a *null-terminated character array*. The *null string* is a character array containing only the null character. A **NULL** pointer is the value that is obtained by casting **0** into a pointer. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error. **NULL** is defined as **0** in **<stdio.h>**; the user can include an appropriate definition if not using **<stdio.h>**.

DESCRIPTION OF SYSTEMCALLS

Upon completion, a system call returns a value. If the system call fails for some reason, the value returned is −1, and additional error information is stored elsewhere. If, on the other hand, the system call succeeds, a value that is generally (but, unfortunately, not always) different from −1 is returned.

If a system call fails, two additional integers containing error information are stored in the program. In order to access these error codes, the programmer should include one or both of the following declarations in the C program:

extern int errno;

extern int smoserr;

The variable *errno* will, upon execution of an unsuccessful system call, contain the so-called *System V error code*. The variable *smoserr* will, upon execution of an unsuccessful system call, contain the so-called *SMOS error code*.

Neither of these error codes is changed when a successful system call is executed, so they should only be inspected when an unsuccessful system call has been detected.

The system call descriptions list most (but not necessarily all) of the System V error codes that are likely to appear during the execution of the system call. In the **<errno.h>** header file, the symbolic names for the System V errors codes are defined.

The SMOS error code is included mainly for compatibility with older versions of the Supermax Operating System. Normally, programmers should not use the SMOS error code, although it does frequently give a more precise description of the error than the System V error code does. In the **<smoserr.h>** header file, the symbolic names for the System V errors are defined. This header file should not be included together with the **<errno.h>** header file, because some of the symbolic names are redefined.

The following is a list of all the available System V error codes and the most common reason for their appearance:

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
No process can be found corresponding to that specified by *pid* in *kill*(2) or *ptrace*(2).

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 has occurred. 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, a tape drive is not on-line or no disk pack is loaded on a drive.

7 **E2BIG** Arg list too long
An argument list longer than 5,120 bytes is presented to a member of the *exec*(2) family.

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*(4)].

9 **EBADF** Bad file number
Either a file descriptor refers to no open file, or a *read*(2) [respectively, *write*(2)] request is made to a file which is open only for writing (respectively, reading).

10 **ECHILD** No child processes
A *wait* was executed by a process that had no existing or unwaited-for child processes.

11 **EAGAIN** No more processes
A *fork* failed because the system's process table is full or the user is not allowed to create any more processes. Or a system call failed because of insufficient memory or swap space.

12 **ENOMEM** Not enough space
During an *exec*(2), *brk*(2), or *sbrk*(2), a program asks for more space than the system is able to supply. This may not be a temporary condition; the maximum space size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during a *fork*(2). If this error occurs on a resource associated with Remote File Sharing (RFS), it indicates a memory depletion wich may be temporary, dependent on system activity at the time the call was invoked.

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 use an argument of a system call.

15 **ENOTBLK** Block device required
A non-block file was mentioned where a block device was required, e.g., in *mount*(2).

16 **EBUSY** Device or resource busy
An attempt was made to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled. The device or resource is currently unavailable.

17 **EEXIST** File exists
An existing file was mentioned in an inappropriate context, e.g., *link*(2).

18 **EXDEV** Cross-device link
A 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 prefix or as an argument to *chdir*(2).

21 **EISDIR** Is a directory
An attempt was made to write on a directory.

22 **EINVAL** Invalid argument
Some invalid argument (e.g., dismounting a non-mounted device; mentioning an undefined signal in *signal*(2) or *kill*(2); reading or writing a file for which *lseek*(2) has generated a negative pointer). Also set by the math functions described in the (3M) entries of this manual.

23 **ENFILE** File table overflow
The system file table is full, and temporarily no more *opens* can be accepted.

24 **EMFILE** Too many open files
No process may have more than NOFILES (default 20) descriptors open at a time.

25 **ENOTTY** Not a character device (or) Not a typewriter
An attempt was made to *ioctl*(2) a file that is not a special character device.

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

27 **EFBIG** File too large
The size of a file exceeded the maximum file size or ULIMIT [see *ulimit*(2)].

28 **ENOSPC** No space left on device
During a *write*(2) to an ordinary file, there is no free space left on the device. In *fcntl*(2), the setting or removing of record locks on a file cannot be accomplished because there are no more record entries left on the system.

29 **ESPIPE** Illegal seek
An *lseek*(2) was issued to a pipe.

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 the maximum number of links (1000) to a file.

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

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

34 **ERANGE** Result too large
The value of a function in the math package (3M) is not representable within machine precision.

35 **ENOMSG** No message of desired type
An attempt was made to receive a message of a type that does not exist on the specified message queue [see *msgop*(2)].

36 **EIDRM** Identifier removed
This error is returned to processes that resume execution due to the removal of an identifier from the file system's name space [see *msgctl*(2), *semctl*(2), and *shmctl*(2)].

37-44 Reserved numbers

45 **EDEADLK** Deadlock
A deadlock situation was detected and avoided. This error pertains to file and record locking.

46 **ENOLCK** No lock
In *fcntl*(2) the setting or removing of record locks on a file cannot be accomplished because there are no more record entries left on the system.

60 **ENOSTR** Not a stream
A *putmsg*(2) or *getmsg*(2) system call was attempted on a file descriptor that is not a STREAMS device.

62 **ETIME** Stream ioctl timeout
The timer set for a STREAMS *ioctl*(2) call has expired. The cause of this error is device specific and could indicate either a hardware or software failure, or perhaps a timeout value that is too short for the specific operation. The status of the *ioctl*(2) operation is indeterminate.

63 **ENOSR** No stream resources
During a STREAMS *open*(2), either no STREAMS queues or no STREAMS head data structures were available.

64 – 70 These errors are Remote File Sharing (RFS) specific.

71 **EPROTO** Protocol error
Some protocol error occurred. This error is device specific, but is generally not related to a hardware failure.

74 **EMULTIHOP** This error is RFS specific

77 **EBADMSG** Bad message
During a *read*(2), *getmsg*(2), or *ioctl*(2) I_RECVFD system call to a STREAMS device, something has come to the head of the queue that can't be processed. That something depends on the system call:

read(2)	–	control information or a passed file descriptor.
getmsg(2)	–	passed file descriptor.
ioctl(2)	–	control or data information.

101 **EWOULDBLOCK** Operation would block
An operation that would cause a process to block was attempted on an object in non-blocking mode (see *fcntl*(2)).

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

103 **EALREADY** Operation already in progress
An operation was attempted on a non-blocking object that already had an operation in progress

104 **ENOTSOCK** Socket operation on non-socket
Self-explanatory.

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

106 **EMSGSIZE** Message too long
A message sent on a socket was larger than the internal message buffer or some other network limit.

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

108 **ENOPROTOOPT** Option not supported by protocol
A bad option or level was specified in a *getsockopt*(3N) or getsockopt call.

109 **EPROTONOSUPPORT** Protocol not supported
The support for the socket type has not been configured into the system or no implementation for it exists.

110 **ESOCKTNOSUPPORT** Socket type unsupported
The support for the socket type has not been configured into the system or no implementation for it exists.

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

112 **EPRNOSUPPORT** Protocol family unsupported
The protocol family has not been configured into the system or no implementation for it exists.

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

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

115 **EADDRNOTAVAIL** Cannot assign requested address
Normally results from an attempt to create a socket with an address not on this machine.

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

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

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

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

120 **ECONNRESET** Connection reset by peer
A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or a reboot.

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

122 **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 when already connected.

123 **ENOTCONN** Socket is unconnected
A request to send or receive data was disallowed because the socket is not connected and (when sending on a datagram socket) no address was supplied.

124 **ESHUTDOWN** Cannot 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.

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

127 **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 that is inactive on the foreign host.

128 **EHOSTDOWN** Host is down
A socket operation failed because the destination host was down.

129 **EHOSTUNREACH** Host is unreachable
A socket operation was attempted to an unreachable host.

Note that shared libraries are currently not implemented in the Supermax Operating System.

The following is a list of all the available SMOS error codes and a short description of the most common reason for their appearance:

1	EDATFUL	OS data area (item area) is full
2	EPRIVIO	Process must be super-user
3	EBADADDR	Bad address
4	EBADDIR	Bad system call number
5	ENOTIMP	Facility not yet implemented
7	EBADPARM	Bad value of argument to system call
50	EPARNX	Partition does not exist
51	EPARAX	Partition already exists
52	ESEGUSE	Segment in use
53	EILSEGNO	Illegal segment number
54	EPARNATT	Partition not attached
55	EPARLONG	Partition too long
56	ENOMEM	No memory

57	EASEGUSE	All segments in use
58	EMAXPAR	All partition descriptors allocated
102	ENOASN	No Address Space Number available
103	EBADLM	Bad load module structure
104	EBADSER	Bad serial number
106	EPROCNX	Process does not exist
110	EPROCABO	Process is being aborted
111	ERESUME	Process was resumed by another process
112	ENOTSUSP	Process is not suspended
113	EMAXPNO	The max. number of local process descriptors exist
114	EDEADPNX	There is no dead process
116	EBADEXNO	Bad signal number
118	ESIGNAL	A signal caused the system call to abort
119	ESTSHORT	The stack is too short to hold parameters
120	ESYSPR	The process is a system process
121	EMAXTD	The maximum number of text descriptors exist
126	EBPIPE	Write on broken pipe
127	EMAXPG	The max. number of global process descriptors exist
128	EMAXATTENT	No more attentions to a specific terminal allowed
129	ENOTMCU	The hardware unit is not an MCU
150	EPROTO	Protocol error
151	EMAXSERVE	Maximum number of RFS servers exist
152	EADV	Advertise error
153	EAADV	Already advertised
154	EMAXADV	Advertise table is full
155	EMAXRCVD	Maximum number of receive descriptors exist

156	EMAXSNDD	Maximum number of send descriptors exist
157	ENONET	Machine is not on the network
160	ECOMM	Communication error on send
161	ENOLINK	The link has been severed
162	ERFS	Remote error, inspect *errno*
163	EMULTIHOP	Multihop attempted
164	ESRMNT	Server mount error
200	EBADACC	File not open for this access mode
201	EBUFLONG	Buffer is too long
203	EILDEVIC	Illegal device
204	EUNITAX	File already exists
205	EUNITNX	File does not exist
206	EILMODE	Illegal access mode
207	EACCVIO	Access right violation
208	ETIMEOUT	I/O operation time out
209	EOPEN	File is already open
210	ENOTOPEN	File is not open
211	EILOP	Illegal operation on specified file
212	EILPOSM	Illegal position mode
213	EILBUFL	Illegal buffer length
214	EEXCDDSK	Transfer exceeds disk
215	ENMOUNT	Disk not mounted
216	EAMOUNT	Disk already mounted
217	EOPENFIL	Files are open on the disk
219	EMAXMOUNT	Mount table is full
220	EISDI	The file is already in direct input mode
221	EISNTDI	The file has not been put in direct input
222	ENREADY	Disk not ready
223	EHARD	Hard error on disk
224	EWRPROT	Disk write protected
225	EILSECT	Illegal sector number
226	ENODATA	No data (for no delay io)
228	EFULLLOC	The lock table is full

229	EBADPOS	Bad position on file
230	ELUSED	The byte range is already locked
232	EILSIZE	Illegal file size or file buffer size
233	EMAXIO	The maximum number of files are open
234	ENOTSIOC	The hardware unit is not a SIOC
235	EDISK	Internal DIOC error
236	ENOSDISK	No sub-disks defined on physical disk
237	EMAXL1	The maximum number of file openings in use
238	ENOREAD	No one is reading from FIFO
239	EILSEEK	Illegal seek on file
243	EFULLDSK	The disk is full
244	EILVARI	Illegal variable record
250	EILTYPE	Illegal file type
254	EMAXL2	Full file table
258	EDEADLK	Byte locking deadlock detected
259	EILFNAM	Illegal file name
262	ENOINO	No i-node available
264	EOLFIL	Outside legal file
266	EMXNLINK	More than 1000 links to a file
267	EMNTMISM	Superblock inconsistency
268	EILMNT	The disk does not contain a file system
270	ESTREAM	Generic streams error, inspect *errno*
271	EMAXMUX	No multiplexer links available
272	EMAXBUF	No streams buffer available
273	EMAXSTREV	No streams events available
274	EMAXQ	No queues available
275	EBADMSG	Trying to read unreadable message
280	EMAXMSG	The max. number of message queues exist
281	EMAXSEM	The max. number of semaphore identifiers exist
282	EMAXSHM	The max. number of shared memory identifiers exist

283	EMSGAX	Message queue already exists
284	EMSGNX	Message queue does not exists
285	EMSQREM	Message queue has been removed
286	ESEMAX	Semaphore identifier already exists
287	ESEMNX	Semaphore identifier does not exist
288	ESEMREM	Semaphore identifier is removed
289	ESEMVBIG	Semaphore value is too big
290	EUNDOFULL	The undo table for the process is full
291	ESEMVAL	The semaphore value would cause the process to wait
292	ENOMSG	No message on queue
293	EMSQFULL	Message queue is full

DEFINITIONS

Process ID

Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 1 to 32,767.

Parent Process ID

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

Process Group ID

Each active process is a member of a process group that is identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes [see *kill*(2)].

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 terminate a group of related processes upon termination of one of the processes in the group [see *exit*(2), *setpgrp*(2), and *signal*(2)].

Real User ID and Real Group ID

Each user allowed on the system is identified by a positive integer (0 to 65535) called a real user ID.

Each user is also a member of a group. The group is identified by a positive integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process.

Effective User ID and Effective Group ID

An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process's real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group ID bit set [see *exec*(2)].

Super-user

A process is recognized as a *super-user* process and is granted special privileges, such as immunity from file permissions, if its effective user ID is 0.

Special Processes

The processes with a process ID less than 80 are special processes. They are used by the operating system itself. The process with process ID 1 is the initialization process (f2init). This process is the ancestor of every other process in the system and is used to control the process structure.

File Descriptor

A file descriptor is a small integer used to do I/O on a file. The value of a file descriptor is from 0 to (OPEN-MAX − 1). A process may have no more than OPEN-MAX file descriptors open simultaneously. The value of OPEN-MAX is a system configuration parameter set by the *chhw*(1M) program. A file descriptor is returned by system calls such as *open*(2), or *pipe*(2). The file descriptor is used as an argument by calls such as *read*(2), *write*(2), *ioctl*(2), and *close*(2).

File Name

Names consisting of 1 to 14 characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding \0 (null) and the ASCII code for / (slash).

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 [see *sh*(1)]. Although permitted, the use of unprintable characters in file names should be avoided.

Path Name and Path Prefix

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.

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.

Unless specifically stated otherwise, the null path name is treated as if it named a non-existent file.

Directory

Directory entries are called links. By convention, a directory contains at least two links, . and .., referred to as *dot* and *dot-dot* respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory

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

File Access Permissions

Read, write, and execute/search permissions on a file are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user.

The effective user ID of the process matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process matches the group of the file and the appropriate access bit of the "group" portion (0070) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process does not match the group ID of the file, and the appropriate access bit of the "other" portion (0007) of the file mode is set.

Otherwise, the corresponding permissions are denied.

Message Queue Identifier

A message queue identifier (msqid) is a unique positive integer created by a *msgget*(2) system call. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and is described in *Programmer's Guide*, Chapter 8.

Semaphore Identifier

A semaphore identifier (semid) is a unique positive integer created by a *semget*(2) system call. Each semid has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and is described in *Programmer's Guide*, Chapter 8.

Shared Memory Identifier
A shared memory identifier (shmid) is a unique positive integer created by a *shmget*(2) system call. Each shmid has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. (Note that these shared memory segments must be explicitly removed by the user after the last reference to them is removed). The data structure is referred to as *shmid_ds* and is described in *Programmer's Guide*, Chapter 8.

STREAMS
are a set of kernel mechanisms that support the development of network services and data communication *drivers*. It defines interface standards for character input/output within the kernel and between the kernel and user level processes. The STREAMS mechanism is composed of utility routines, kernel facilities and a set of data structures.

Stream
A stream is a full-duplex data path within the kernel between a user process and driver routines. The primary components are a *stream head*, a *driver* and zero or more *modules* between the *stream head* and *driver*. A *stream* is analogous to a Shell pipeline except that data flow and processing are bidirectional.

Stream Head.
In a *stream*, the *stream head* is the end of the *stream* that provides the interface between the *stream* and a user process. The principle functions of the *stream head* are processing STREAMS-related system calls, and passing data and information between a user process and the *stream*.

Driver
In a *stream*, the *driver* provides the interface between peripheral hardware and the *stream*. A *driver* can also be a pseudo-*driver*, such as a *multiplexor*, which is not associated with a hardware device.

Module
A module is an entity containing processing routines for input and output data. It always exists in the middle of a *stream*, between the stream's head and a *driver*. A *module* is the STREAMS counterpart to the commands in a Shell pipeline except that a module contains a pair of functions which allow independent bidirectional (*downstream* and *upstream*) data flow and processing.

Downstream
In a *stream*, the direction from *stream head* to *driver*.

Upstream
In a *stream*, the direction from *driver* to *stream head*.

Message
In a *stream*, one or more blocks of data or information, with associated STREAMS control structures. *Messages* can be of several defined types, which identify the *message* contents. *Messages* are the only means of transferring data and communicating within a *stream*.

Message Queue
In a *stream*, a linked list of *messages* awaiting processing by a *module* or *driver*.

Read Queue
In a *stream*, the *message queue* in a *module* or *driver* containing *messages* moving *upstream*.

Write Queue
In a *stream*, the *message queue* in a *module* or *driver* containing *messages* moving *downstream*.

Multiplexor
A multiplexor is a driver that allows *streams* associated with several user processes to be connected to a single *driver*, or several *drivers* to be connected to a single user process.

STREAMS does not provide a general multiplexing *driver*, but does provide the facilities for constructing them, and for connecting multiplexed configurations of *streams*.

FILES

/lib/libc.a
/lib/libm.a

SEE ALSO

ar(1), cc(1), ld(1), lint(1), nm(1), stdio(3S), math(5).

DIAGNOSTICS

Functions in the C and Math Libraries (3C and 3M) may return the conventional values **0** or ±**HUGE_VAL** (the largest-magnitude single-precision floating-point numbers; **HUGE_VAL** is defined in the <*math.h*> header file) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable *errno* is set to the value EDOM or ERANGE.

WARNING

Many of the functions in the libraries call and/or refer to other functions and external variables described in this section. If a program inadvertently defines a function or external variable with the same name, the presumed library version of the function or external variable may not be loaded.

NAME

a64l, l64a – convert between long integer and base-64 ASCII string

SYNOPSIS

long a64l (s)
char *s;

char *l64a (l)
long l;

DESCRIPTION

These functions are used to maintain numbers stored in *base-64* ASCII characters. This is a notation by which long integers can be represented by up to six characters; each character represents a "digit" in a radix-64 notation.

The characters used to represent "digits" are **.** for 0, **/** for 1, **0** through **9** for 2–11, **A** through **Z** for 12–37, and **a** through **z** for 38–63.

a64l takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. If the string pointed to by *s* contains more than six characters, *a64l* will use the first six.

a64l scans the character string from left to right, decoding each character as a 6 bit Radix 64 number.

l64a takes a **long** argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, *l64a* returns a pointer to a null string.

CAVEAT

The value returned by *l64a* is a pointer into a static buffer, the contents of which are overwritten by each call.

This page is intentionally left blank

NAME

abort – generate an abnormal program termination

SYNOPSIS

#include <stdlib.h>

void abort ( )

DESCRIPTION

abort does the work of *exit*(2), but instead of just exiting, *abort* causes **SIGABRT** to be sent to the calling process. If **SIGABRT** is neither caught nor ignored, all *stdio*(3S) streams are flushed prior to the signal being sent, and a core dump results.

abort returns the value of the *kill*(2) system call.

SEE ALSO

sdb(1), exit(2), kill(2), signal(2).

DIAGNOSTICS

If **SIGABRT** is neither caught nor ignored, and the current directory is writable, a core dump is produced and the message "Abort (coredump)" is written by the shell.

SIGABRT is not intended to be caught.

This page is intentionally left blank

NAME

abs − return integer absolute value

SYNOPSIS

```
#include <stdlib.h>
int abs (i)
 int i;
```

DESCRIPTION

abs returns the absolute value of its integer operand.

SEE ALSO

floor(3M).

CAVEAT

In two's-complement representation, the absolute value of the negative integer with largest magnitude is undefined.

This page is intentionally left blank

NAME

access – determine accessibility of a file

SYNOPSIS

#include <unistd.h>

```
int access (path, amode)
 char * path;
 int amode;
```

DESCRIPTION

path points to a path name naming a file. *access* checks the named file for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. The bit pattern contained in *amode* is constructed from symbolic constants defined by the **<unistd.h>** header file. They are as follows:

Name	*Value*	*Description*
R_OK	04	test for *read* permission
W_OK	02	test for *write* permission
X_OK	01	test for *execute (search)* permission
F_OK	00	test for existence of file

amode is either the logical OR of the values of the symbolic constants for R_OK, W_OK, and X_OK or is the value of the symbolic constant F_OK.

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

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

[EACCESS] Permission bits of the file mode do not permit the requested access.

[EFAULT]	*path* points outside the allocated address space for the process.
[EINTR]	A signal was caught during the *access* system call.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX}, or the length of a *path* component exceeds {NAME_MAX} while _POSIX_NO_TRUNC is in effect.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	Read, write, or execute (search) permission is requested for a null path name.
[ENOENT]	The named file does not exist.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EROFS]	Write access is requested for a file on a read-only file system.

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 permissions checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.

SEE ALSO

chmod(2), stat(2).

DIAGNOSTICS

If the requested access is permitted, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

acct – enable or disable process accounting

SYNOPSIS

int acct (path)
char * path;

DESCRIPTION

acct is used to enable or disable the system process accounting routine. If the routine is enabled, an accounting record will be written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a signal [see *exit*(2) and *signal*(2)]. The effective user ID of the calling process must be superuser to use this call.

path points to a pathname naming the accounting file. The accounting file format is given in *acct*(4).

The accounting routine is enabled if *path* is non-zero and no errors occur during the system call. It is disabled if *path* is zero and no errors occur during the system call.

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

[EACCESS] The file named *path* is not an ordinary file.

[EBUSY] An attempt is being made to enable accounting when it is already enabled.

[EFAULT] *path* points to an illegal address.

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

[ENOENT] One or more components of the accounting file pathname do not exist.

[EPERM] The effective user of the calling process is not superuser.

[EROFS] The named file resides on a read-only file system.

SEE ALSO

exit(2), signal(2), acct(4).

DIAGNOSTICS

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

NAME

alarm – set a process alarm clock

SYNOPSIS

unsigned alarm (sec)
unsigned sec;

DESCRIPTION

alarm instructs the alarm clock of the calling process to send the signal **SIGALRM** to the calling process after the number of real time seconds specified by *sec* have elapsed [see *signal*(2)].

Alarm requests are not stacked; successive calls reset the alarm clock of the calling process.

If *sec* is 0, any previously made alarm request is canceled.

SEE ALSO

pause(2), signal(2), sigpause(2), sigset(2).

DIAGNOSTICS

alarm returns the amount of time previously remaining in the alarm clock of the calling process.

This page is intentionally left blank

NAME

amsgop – asynchronous message operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <aio.h>

int amsgsnd(resbuf, msqid, msgp, msgsz, msgflg)
  struct a_res *resbuf;
  int msqid;
  struct mymsg *msgp;
  int msgsz, msgflg;

int amsgrcv(resbuf,msqid,msgp,msgsz,msgtyp,msgflg)
  struct a_res *resbuf;
  int msqid;
  struct mymsg *msgp;
  int msgsz;
  long msgtyp;
  int msgflg;
```

DESCRIPTION

amsgsnd and *amsgrcv* initiate an asynchronous attempt to send or receive a message via a message queue. The operation performed by *amsgsnd* and *amsgrcv* is identical to that performed by *msgsnd* and *msgrcv*, respectively, and the reader should consult the manual page on *msgop*(2) for a description of the arguments, *msqid*, *msgp*, *msgsz*, *msgtyp*, and *msgflg*, as well as for a detailed description of the operation performed.

However, in contrast to *msgsnd* and *msgrcv*, the operation performed by *amsgsnd* and *amsgrcv* is asynchronous, that is, the operating system returns immediately to the calling process, allowing this process to perform other computations while the message operation is in progress. The details are as follows:

When *amsgsnd* or *amsgrcv* is called, a child process of the calling process is created. This child process performs the actual message operation, while the parent process continues execution. Once the message operation is completed, the child process dies.

The parent process may inspect the result of the message operation through the *waitx*(2X) routines. These routines work as they always do, inspecting the state of dead child processes; if, however, the child process is an asynchronous message operation process, the *wait*(2) routine will cause status information to be returned in the structure pointed to by the *resbuf* argument of the *amsgsnd* or *amsgrcv* call.

The *a_res* structure is defined in the **<aio.h>** header file and contains the following fields that will be set by the *wait*(2) routine:

```
long a_type;     /* set to the symbolic constant
                    AMSGSND or AMSGRCV */
long a_res;      /* the return value from
                    msgsnd or msgrcv */
long a_smoserr;  /* smoserr error code of
                    message operation */
long a_errno;    /* errno error code of
                    message operation */
```

The *errno* error code of the message operation may also be found as the exit code of the child process.

The name of the child process returned by the *waitx* routine [see *wait*(2)] will be the name of the parent process with the character 'A' appended.

No more than 256 bytes may be transferred at a time.

amsgsnd and *amsgrcv* will fail and no child process will be created if one or more of the follwing are true:

[EINVAL]	*msqid* is not a valid message queue identifier; or the value of *msgsz* is less than 0 or greater than 256.
[EACCES]	The *msg_perm.mode* field of the data structure associated with the message queue identifier denies the necessary permission.
[EFAULT]	*msgp* or *resbuf* points to an illegal address.
[EAGAIN]	The maximum number of processes on the MCU or on the entire computer would be exceeded.

SEE ALSO

msgop(2), wait(2), waitx(2X).

DIAGNOSTICS

If the starting of the child process is successful, *amsgsnd* and *amsgrcv* will return the process ID of the child process; otherwise they will return -1, no child process will be started, and *errno* will indicate the error.

Note: If *amsgsnd* or *amsgrcv* do not return -1, the message operation may still fail. In this case the error is indicated in the *a_res* structure.

NOTE

The program must be loaded with the library **libdde.a**.

This page is intentionally left blank

NAME

aread – asynchronous read

SYNOPSIS

```
#include <aio.h>

int aread (resbuf, fildes, buf, nbyte)
  struct a_res *resbuf;
  int fildes;
  char *buf;
  unsigned nbyte;
```

DESCRIPTION

aread initiates an asynchronous attempt to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*. The operation performed by *aread* is identical to that performed by *read* and the reader should consult the manual page on this routine for a description of the arguments, *fildes*, *buf*, and *nbyte*, as well as for a detailed description of the operation performed.

However, in contrast to *read*, the operation performed by *aread* is asynchronous, that is, the operating system returns immediately to the calling process, allowing this process to perform other computations while the read operation is in progress. The details are as follows:

When *aread* is called, a child process of the calling process is created. This child process performs the actual read operation, while the parent process continues execution. Once the read operation is completed, the child process dies.

The parent process may inspect the result of the read operation through the *wait*(2) routines. These routines work as they always do, inspecting the state of dead child processes; if, however, the child process is an asynchronous I/O operation process, the *wait*(2) routines will cause status information to be returned in the structure pointed to by the *resbuf* argument of the *aread* call.

The *a_res* structure is defined in the **<aio.h>** header file and contains the following fields that will be set by the *wait*(2) routines:

```
long a_type;     /* set to the symbolic constant
                    AREAD */
long a_res;      /* the number of bytes actually read,
                    or -1 for error */
long a_smoserr;  /* smoserr error code of
                    read operation */
long a_errno;    /* errno error code of
                    read operation */
long a_curpos;   /* cursor position at end of
                    terminal input */
long a_funkey;   /* key that terminated
                    terminal input */
```

The *errno* error code of the read operation may also be found as the exit code of the child process.

The name of the child process returned by the *waitx* routine [see *wait*(2)] will be the name of the parent process with the character 'A' appended.

No more than 256 bytes may be read at a time.

aread will fail and no child process will be created if one or more of the follwing are true:

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

[EINVAL] *nbyte* is greater than 256.

[EFAULT] *buf* or *resbuf* points to an illegal address.

[EAGAIN] The maximum number of processes on the MCU or on the entire computer would be exceeded.

SEE ALSO

read(2), wait(2).

DIAGNOSTICS

If the starting of the reading child process is successful, *aread* will return the process ID of the child process; otherwise it will return −1, no child process will be started, and *errno* will indicate the error.

Note: If *aread* does not return −1, the read operation may still fail. In this case the error is indicated in the *a_res* structure.

NOTE

The program must be loaded with the library **libdde.a**.

This page is intentionally left blank

NAME

assert – verify program assertion

SYNOPSIS

#include <assert.h>

assert (expression)
int expression;

DESCRIPTION

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false (zero), *assert* prints

"Assertion failed: *expression*, file *xyz*, line *nnn*"

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the *assert* statement.

Compiling with the preprocessor option **–DNDEBUG** [see *cpp* (1)], or with the preprocessor control statement "**#define** NDEBUG" ahead of the "**#include** <assert.h>" statement, will stop assertions from being compiled into the program.

SEE ALSO

cpp(1), abort(3C).

CAVEAT

Since *assert* is implemented as a macro, the *expression* may not contain any string literals.

This page is intentionally left blank

NAME

asuspend – asynchronous suspend

SYNOPSIS

#include <aio.h>

int asuspend (resbuf, time)
struct a_res *resbuf;
long time;

DESCRIPTION

asuspend performs an asynchronous suspension for *time* milliseconds (with a resolution of 40 milliseconds).

When *asuspend* is called, a child process of the calling process is created. This child suspends itself for the specified time, while the parent process continues execution. Once the suspend time has expired, the child process dies. The operation actually performed by the child process is a *suspend*(2) system call with a *pid* argument of −1.

The parent process may inspect the result of the suspend operation through the *wait*(2) routines. These routines work as they always do, inspecting the state of dead child processes; if, however, the child process is an asynchronous suspension, the *wait*(2) routines will cause status information to be returned in the structure pointed to by the *resbuf* argument of the *asuspend* call.

The *a_res* structure is defined in the **<aio.h>** header file and contains the following fields that will be set by the *wait*(2) routines:

```
long a_type;     /* set to the symbolic constant
                    ASUSPEND */
long a_res;      /* the return value from
                    suspend(2X) */
long a_smoserr;  /* smoserr error code of
                    suspend operation */
long a_errno;    /* errno error code of
                    suspend operation */
```

The *errno* error code of the suspend operation may also be found as the exit code of the child process.

The name of the child process returned by the *waitx* routine [see *waitx*(2X)] will be the name of the parent process with the character 'A' appended.

The *asuspend* function can be used in connection with other asynchronous operations to ensure that a *wait*(2) routine will return within a specified amount of time. The *alarm*(2) routine can also be used for this purpose, but *asuspend* has a finer resolution.

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

[EAGAIN] The maximum number of processes on the MCU or on the entire computer would be exceeded.

[EFAULT] *resbuf* points to an illegal address.

SEE ALSO

suspend(2X), waitx(2X).

DIAGNOSTICS

If the starting of the suspending child process is successful, *asuspend* will return the process ID of the child process; otherwise it will return −1, no child process will be started, and *errno* will indicate the error.

NOTE

The program must be loaded with the library **libdde.a**.

NAME

atexit – add program termination routine

SYNOPSIS

#include <stdlib.h>

int atexit (func)

void *func();

DESCRIPTION

atexit adds the function *func* to a list of functions to be called without arguments on normal termination of the program. Normal termination occurs by either a call to the *exit* system call or a return from *main*. At most 32 functions may be registered by *atexit*; the functions will be called in the reverse order of their registration.

atexit returns 0 of the registration succeeds, nonzero if it fails.

SEE ALSO

exit(2).

This page is intentionally left blank

NAME

awrite – asynchronous write

SYNOPSIS

```
#include <aio.h>

int awrite (resbuf, fildes, buf, nbyte)
 struct a_res * resbuf;
 int fildes;
 char * buf;
 unsigned nbyte;
```

DESCRIPTION

awrite initiates an asynchronous attempt to write *nbyte* bytes to the file associated with *fildes* from the buffer pointed to by *buf*. The operation performed by *awrite* is identical to that performed by *write* and the reader should consult the manual page on this routine for a description of the arguments, *fildes*, *buf*, and *nbyte*, as well as for a detailed description of the operation performed.

However, in contrast to *write*, the operation performed by *awrite* is asynchronous, that is, the operating system returns immediately to the calling process, allowing this process to perform other computations while the write operation is in progress. The details are as follows:

When *awrite* is called, a child process of the calling process is created. This child process performs the actual write operation, while the parent process continues execution. Once the write operation is completed, the child process dies.

The parent process may inspect the result of the write operation through the *waitx*(2X) routines. These routines work as they always do, inspecting the state of dead child processes; if, however, the child process is an asynchronous I/O operation process, the *wait*(2) routines will cause status information to be returned in the structure pointed to by the *resbuf* argument of the *awrite* call.

The *a_res* structure is defined in the **<aio.h>** header file and contains the following fields that will be set by the *wait*(2) routines:

```
long a_type;     /* set to the symbolic constant
                    AWRITE */
long a_res;      /* the number of bytes actually written,
                    or -1 for error */
long a_smoserr;  /* smoserr error code of
                    write operation */
long a_errno;    /* errno error code of
                    write operation */
```

The *errno* error code of the write operation may also be found as the exit code of the child process.

The name of the child process returned by the *waitx* routine [see *wait*(2)] will be the name of the parent process with the character 'A' appended.

No more than 256 bytes may be written at a time.

awrite will fail and no child process will be created if one or more of the follwing are true:

[EAGAIN]	The maximum number of processes on the MCU or on the entire computer would be exceeded.
[EBADF]	*fildes* is not a valid file descriptor open for writing.
[EFAULT]	*buf* or *resbuf* points to an illegal address.
[EINVAL]	*nbyte* is greater than 256.

SEE ALSO

wait(2), waitx(2X), write(2).

DIAGNOSTICS

If the starting of the writing child process is successful, *awrite* will return the process ID of the child process; otherwise it will return −1, no child process will be started, and *errno* will indicate the error.

Note: If *awrite* does not return -1, the write operation may still fail. In this case the error is indicated in the *a_res* structure.

NOTE

The program must be loaded with the library **libdde.a**.

This page is intentionally left blank

NAME

bessel: 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)
  int n;
  double x;

double y0 (x)
  double x;

double y1 (x)
  double x;

double yn (n, x)
  int n;
  double x;
```

DESCRIPTION

j0 and *j1* return Bessel functions of x of the first kind of orders 0 and 1 respectively. *jn* returns the Bessel function of x of the first kind of order n.

y0 and *y1* return Bessel functions of x of the second kind of orders 0 and 1 respectively. *yn* returns the Bessel function of x of the second kind of order n.

DIAGNOSTICS

Non-positive arguments cause *y0*, *y1* and *yn* to return the value **–HUGE** and to set *errno* to **EDOM**.

If x is NaN, NaN is returned and *errno* is set to **EDOM**.

Arguments too large in magnitude cause *j0*, *j1*, *y0* and *y1* to return zero and to set *errno* to **ERANGE**.

This page is intentionally left blank

NAME

brk, sbrk – change data segment space allocation

SYNOPSIS

int brk (endds)
char ∗ endds;

char ∗ sbrk (incr)
int incr;

DESCRIPTION

brk and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment [see *exec*(2)]. The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. Newly allocated space is set to zero. If, however, the same memory space is reallocated to the same process its contents are undefined.

brk sets the break value to *endds* and changes the allocated space accordingly.

sbrk adds *incr* bytes to the break value and changes the allocated space accordingly. *Incr* can be negative, in which case the amount of allocated space is decreased.

brk and *sbrk* will fail without making any change in the allocated space if one or more of the following are true:

[EINVAL] The address of the allocated memory would conflict with the address of an already allocated shared memory segment (a memory partition).

[ENOMEM] Such a change would result in more space being allocated than is allowed by the system-imposed maximum process size [see *ulimit*(2)].

[ENOSPC] The maximum number of available memory partition descriptors would be exceeded.

SEE ALSO

exec(2), shmop(2), ulimit(2), end(3C), malloc(3C).

DIAGNOSTICS

Upon successful completion, *brk* returns a value of 0 and *sbrk* returns the old break value. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

bsearch – binary search a sorted table

SYNOPSIS

```
#include <stdlib.h>

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

DESCRIPTION

bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *key* points to a datum instance to be sought in the table. *base* points to the element at the base of the table. *nel* is the number of elements in the table. *compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according to whether the first argument is to be considered less than, equal to, or greater than the second.

EXAMPLE

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

This code fragment reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

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

#define     TABSIZE     1000
```

```
struct node {
            /* these are stored in the table */
      char *string;
      int length;
};
struct node table[TABSIZE];
            /* table to be searched */
      .
      .
{
      struct node *node_ptr, node;
      int node_compare( );
            /* routine to compare 2 nodes */
      char str_space[20];
            /* space to read string into */
      .
      .
      node.string = str_space;
      while (scanf("%s", node.string) != EOF) {
            node_ptr = (struct node *)bsearch
                  ((void *)(&node),
                  (void *)table, TABSIZE,
                  sizeof(struct node),
                  node_compare);
            if (node_ptr != NULL) {
                  (void)printf("string = %20s,
                    length = %d\n",
                        node_ptr->string,
                        node_ptr->length);
            } else {
                  (void)printf("not found: %s\n",
                  node.string);
            }
      }
}
```

```
/*
        This routine compares two nodes based on an
        alphabetical ordering of the string field.
 */
int
node_compare(node1, node2)
void *node1, *node2;
{
        return (strcmp(
                ((struct node *)node1)->string,
                ((struct node *)node2)->string));
}
```

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-void.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although *bsearch* is declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

hsearch(3C), lsearch(3C), qsort(3C), tsearch(3C).

DIAGNOSTICS

A NULL pointer is returned if the key cannot be found in the table.

This page is intentionally left blank

NAME

catgets – read a program message

SYNOPSIS

```
#include <nl_types.h>

char *catgets (catd, set_id, msg_id, s)
nl_catd catd;
int set_id, msg_id;
char *s;
```

DESCRIPTION

catgets attempts to read message *msg_num*, in set *set_num*, from the message catalogue identified by *catd*. *catd* is a catalogue descriptor returned from an earlier call to *catopen*. *s* points to a default message string which will be returned by *catgets* if the identified message catalogue is not currently available.

SEE ALSO

catopen(3C).

DIAGNOSTICS

If the identified message is retrieved successfully, *catgets* returns a pointer to an internal buffer area containing the null terminated message string. If the call is unsuccessful because the message catalogue identified by *catd* is not currently available, a pointer to *s* is returned.

This page is intentionally left blank

NAME

catopen, catclose – open/close a message catalogue

SYNOPSIS

#include <nl_types.h>

nl_catd catopen (name, oflag)
char * name;
int oflag;

int catclose (catd)
nl_catd catd;

DESCRIPTION

catopen opens a message catalogue and returns a catalogue descriptor. *name* specifies the name of the message catalogue to be opened. If *name* contains a '/' then *name* specifies a pathname for the message catalogue. Otherwise, the environment variable NLSPATH is used. If NLSPATH does not exist in the environment, or if a message catalogue cannot be opened in any of the paths specified by NLSPATH, then the default path is used (see *nl_types*(5)).

The names of message catalogues, and their location in the filestore, can vary from one system to another. Individual applications can choose to name or locate message catalogues according to their own special needs. A mechanism is therefore required to specify where the catalogue resides.

The NLSPATH variable provides both the location of message catalogues, in the form of a search path, and the naming conventions associated with message catalogue files. For example:

NLSPATH=/usr/lib/locale/%L/%N.cat:/usr/lib/locale/%N/%L

The metacharacter **%** introduces a substitution field, where **%L** substitutes the current setting of the LANG environment variable (see following section), and **%N** substitutes the value of the *name* parameter passed to *catopen*. Thus, in the above example, *catopen* will search in /**$**LANG/*name*.cat, then in /*name*/**$**LANG, for the required message catalog.

NLSPATH will normally be set up on a system wide basis (e.g., in /etc/profile) and thus makes the location and naming conventions associated with message catalogues transparent to both programs and users.

The full set of metacharacters is:

- **%N** The value of the name parameter passed to *catopen*.
- **%L** The value of LANG.
- **%l** The value of the language element of LANG.
- **%t** The value of the territory element of LANG.
- **%c** The value of the codeset element of LANG.
- **%%** A single %.

The LANG environment variable provides the ability to specify the users requirements for native languages, local customs and character set, as an ASCII string in the form

LANG=language[_territory[.codeset]]

A user who speaks German as it is spoken in Schwitzerland and has a terminal which operates in ISO 8859/1 codeset, would want the setting of the LANG variable to be

LANG=de_CH.88591

With this setting it should be possible for that user to find any relevant catalogs should they exist.

Should the LANG variable not be set then the value of LC_MESSAGES as returned by *setlocale* is used. If this is NULL then the default path as defined in *nl_types* is used.

oflag is reserved for future use and should be set to 0. The results of setting this field to any other value are undefined.

catclose closes the message catalog identified by *catd*.

SEE ALSO

catgets(3C), setlocale(3C), environ(5), nl_types(5).

DIAGNOSTICS

If successful, *catopen* returns a message catalog descriptor for use on subsequent calls to *catgets* and *catclose*. Otherwise *catopen* returns *(nl_catd)* −1. *catclose* returns 0 if successful, otherwise −1.

This page is intentionally left blank

NAME

chdir – change working directory

SYNOPSIS

int chdir (path)
char *path;

DESCRIPTION

path points to the path name of a directory. *chdir* causes the named directory to become the current working directory, that is the starting point for path searches for path names not beginning with "/" .

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

[EACCES]	Search permission is denied for any component of the path name.
[EFAULT]	*path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *chdir* system call.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX}, or the length of a *path* component exceeds {NAME_MAX} while _POSIX_NO_TRUNC is in effect.
[ENOENT]	The named directory does not exist.
[ENOLINK]	*path* points to a remeote machine and the link to that machine is no longer active.
[ENOTDIR]	A component of the path name is not a directory.

SEE ALSO

chroot(2).

DIAGNOSTICS

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

NAME

chklicense – check if program has license to run

SYNOPSIS

#include <license.h>

int chklicense (stocknumber)
int stocknumber;

DESCRIPTION

chklicense checks if this program is allowed to run under the license conditions which are present on this installation. It is called with the stock number of the calling program, and *chklicense* will fail if one or more of the following are true:

[EACCESS] There are no licenses available for the specified stock number, because they are all in use at this time, or the date has expired.

[EINVAL] *chklicense* has been called before by this process with a different stock number. This may be used to check if the real call to *chklicense* has been patched out.

[ENOEXEC] No licenses have been loaded for the given stock number. (Done by /etc/loadlicense during booting).

When *chklicense* is called the kernel will check if there are any more licenses left for the given stock number. If so, the kernel will register the stock number of the program and when the program terminates the license will be released.

EXAMPLE

The check of the license conditions on the host may look like:

```
if (chklicense(stocknumber) < 0){
   if (errno = = EACCESS){
      perror("Thisprogram");
      fprintf(stderr, "Licenses used up or
                 nonexistent for %d", stocknumber);
      }
```

```
        if (errno = = ENOEXEC){
           perror("Thisprogram");
           fprintf(stderr, "No license for %d", stocknumber);
           }

        if (errono == EINVAL){
           perror("Thisprogram");
           fprintf(stderr, "Chklicense called before with");
           fprintf(stderr, " different stocknumber than
                       %d0, stocknumber);
           }
     exit(1);
     }
     .
     .
     .
```

SEE ALSO

instno(1), license(4).

DIAGNOSTICS

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

NAME

chmod – change mode of file

SYNOPSIS

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

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

DESCRIPTION

path points to a path name naming a file. *chmod* sets the access permission portion of the named file's mode according to the bit pattern contained in *mode*.

Access permission bits are described in **<sys/stat.h>**, and are interpreted as follows:

Name	*Value*	*Description*
S_ISUID	04000	Set user-ID on execution.
S_ISGID	02000	Set group-ID on execution.
S_ENFMT	02000	Enable mandatory file/record locking.
	01000	Reserved.
S_IRUSR	00400	Read by owner.
S_IWUSR	00200	Write by owner.
S_IXUSR	00100	Execute (search if a directory) by owner.
S_IRGRP	00040	Read by group.
S_IWGRP	00020	Write by group.
S_IXGRP	00010	Execute (search) by group.
S_IROTH	00004	Read by others (that is, anyone else).
S_IWOTH	00002	Write by others.
S_IXOTH	00001	Execute (search) by others.

Note that the value of S_ISGID and S_ENFMT have the same value. That particular bit is interpreted as S_ISGID if S_IXGRP is set; it is interpreted as S_ENFMT if S_IXGRP is not set.

The effective user ID of the process must match the owner of the file or be super-user to change the mode of a file.

If the effective user ID of the process is not super-user and the effective group ID of the process does not match the group ID of the file, mode bit S_ISGID is cleared.

chmod will fail and the file mode will be unchanged if one or more of the following are true:

[EACCES]	Search permission is denied on a component of the path prefix.
[EFAULT]	*path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *chmod* system call.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX}, or the length of a *path* component exceeds {NAME_MAX} while _POSIX_NO_TRUNC is in effect.
[ENOENT]	The named file does not exist.
[ENOLINK]	*path*points to a remote machine and the link to that machine is no longer active.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not super-user.
[EROFS]	The named file resides on a read-only file system.

SEE ALSO

chmod(1), chown(2), creat(2), fcntl(2), mknod(2), open(2), read(2), write(2).

DIAGNOSTICS

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

This page is intentionally left blank

NAME

chown – change owner and group of a file
lchown – change owner and group of a symbolic link

SYNOPSIS

#include <sys/types.h>

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

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

DESCRIPTION

With **chown** *path* points to a path name naming a file. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively. If *path* is a symbolic link the owner (group) of the file pointed at will be changed.

lchown is similar to **chown** except for symbolic links where the file containing the link will change owner (group).

Only processes with effective user ID equal to the file owner or super-user may change the ownership of a file.

If **chown** is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

chown will fail and the owner and group of the named file will remain unchanged if one or more of the following are true:

[EACCES]	Search permission is denied on a component of the path prefix.
[EFAULT]	*path* points outside the allocated address space of the process.

[EINTR]	A signal was caught during the *chown* system call.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOENT]	The named file does not exist.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not super-user.
[EROFS]	The named file resides on a read-only file system.

SEE ALSO

chmod(2), chown(1).

DIAGNOSTICS

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

NOTE

The function *lchown* must be loaded with the library **libdde.a**

NAME

chroot – change root directory

SYNOPSIS

int chroot (path)
char * path;

DESCRIPTION

path points to a path name naming a directory. *chroot* causes the named directory to become the root directory, the starting point for path searches for path names beginning with /. The user's working directory is unaffected by the *chroot* system call.

The effective user ID of the process must be super-user to change the root directory.

The **..** entry in the root directory is interpreted to mean the root directory itself. Thus, **..** cannot be used to access files outside the subtree rooted at the root directory.

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

[EACCES]	Search permission is denied for a component of *path*.
[EFAULT]	*path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *chroot* system call.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOENT]	The named directory does not excist.

[ENOLINK] *path* points to a remote machine and the link to that machine is no longer active. [ENOTDIR] Any component of the path name is not a directory.

[EPERM] The effective user ID is not super-user.

SEE ALSO

chdir(2).

DIAGNOSTICS

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

NAME

clock – report CPU time used

SYNOPSIS

#include <time.h>

clock_t clock ( )

DESCRIPTION

clock returns the amount of CPU time (in microseconds) used since the first call to *clock*. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed *wait*(2), *pclose*(3S), or *system*(3S).

The resolution of the clock is 40 milliseconds on Supermax computers.

SEE ALSO

times(2), wait(2), popen(3S), system(3S).

BUGS

The value returned by *clock* is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned will wrap around after accumulating only 2147 seconds of CPU time (about 36 minutes).

This page is intentionally left blank

NAME

close – close a file descriptor

SYNOPSIS

int close (fildes)
int fildes;

DESCRIPTION

fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *close* closes the file descriptor indicated by *fildes*. All outstanding record locks owned by the process (on the file indicated by *fildes*) are removed.

If a STREAMS [see *intro*(2)] file is closed, and the calling process had previously registered to receive a SIGPOLL signal [see *signal*(2) and *sigset*(2)] for events associated with that file [see I_SETSIG in *streamio*(7)], the calling process will be unregistered for events associated with the file. The last *close* for a *stream* causes the *stream* associated with *fildes* to be dismantled. If O_NDELAY is not set and there have been no signals posted for the *stream*, *close* waits up to 15 seconds, for each module and driver, for any output to drain before dismantling the *stream*. If the O_NDELAY flag is set or if there are any pending signals, *close* does not wait for output to drain, and dismantles the *stream* immediately.

The named file is closed unless one or more of the following are true:

[EBADF]	*fildes* is not a valid open file descriptor.
[EINTR]	A signal was caught during the *close* system call.
[ENOLINK]	*fildes* is on a remote machine and the link to that machine is no longer ctive.

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), intro(2), open(2), pipe(2), signal(2), sigset(2), streamio(7).

DIAGNOSTICS

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

NAME

conv: toupper, tolower, _toupper, _tolower, toascii – translate characters

SYNOPSIS

#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;

DESCRIPTION

toupper and *tolower* have as their domain the range of the function *getc*(3S): all values represented in an *unsigned char* and the value of the macro EOF as defined in *stdio.h*. If the argument of *toupper* represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of *tolower* represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

The macros *_toupper* and *_tolower* accomplish the same thing as *toupper* and *tolower*, respectively, but have restricted domains and are faster. *_toupper* requires a lower-case letter as its argument; its result is the corresponding upper-case letter. *_tolower* requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results.

The macro *toascii* yields its argument with all bits turned off that are not part of a standard 7-bit ASCII character; it is intended for compatibility with other systems.

toupper, *tolower*, *_toupper*, and *_tolower* are affected by LC_CTYPE. In the C locale, or in a locale where shift information is not defined, these functions determine the case of characters according to the rules of the ASCII-coded character set. Characters outside the ASCII range of characters are returned unchanged.

SEE ALSO

ctype(3C), getc(3S), setlocale(3C), environ(5).

NAME

creat – create a new file or rewrite an existing one

SYNOPSIS

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

int creat (path, mode)
 char * path;
 mode_t mode;

DESCRIPTION

creat creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by *path*.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the effective user ID, of the process the group ID of the process is set to the effective group ID, of the process and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows:

> All bits set in the process's file mode creation mask are cleared [see *umask*(2)].

Upon successful completion, a write-only file descriptor is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across *exec* system calls [see *fcntl*(2)]. No process may have more than OPEN_MAX files open simultaneously. The value of OPEN_MAX is set by the *chhw*(1M) program. A new file may be created with a mode that forbids writing.

creat fails if one or more of the following are true:

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

[EACCES]	The file does not exist and the directory in which the file is to be created does not permit writing.
[EACCES]	The file exists and write permission is denied.
[EAGAIN]	The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see *chmod*(2)].
[EINTR]	A signal was caught during the *create* system call.
[EISDIR	The named file is an existing directory.
[EFAULT]	*path* points outside the allocated address space of the process.
[EMFILE]	OPEN_MAX file descriptors are currently open.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[ENFILE]	The system file table is full.
[ENOENT]	A component of the path prefix does not exist.
[ENOENT]	The path name is null.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[ENOSPC]	The file system is out of inodes.
[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	The named file resides or would reside on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.

SEE ALSO

chmod(2), close(2), dup(2), fcntl(2), lseek(2), open(2), read(2), umask(2), write(2).

DIAGNOSTICS

Upon successful completion, a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

This page is intentionally left blank

NAME

crypt, setkey, encrypt – generate hashing encryption

SYNOPSIS

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

void setkey (key)
 char *key;

void encrypt (block, ignored)
 char block[64];
 int ignored;
```

DESCRIPTION

crypt is the password encryption function. It is based on a one way hashing encryption algorithm with variations intended (among other things) to frustrate use of hardware implementations of a key search.

key is a user's typed password. *salt* is a two-character string chosen from the set [**a-zA-Z0-9./**]; this string is used to perturb the hashing 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. The first two characters are the salt itself.

The *setkey* and *encrypt* entries provide (rather primitive) access to the actual hashing 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; this gives a 56-bit key which is set into the machine.

This is the key that will be used with the hashing algorithm to encrypt the string *block* with the function *encrypt.*

The argument to the *encrypt* entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by *setkey*. *Ignored* is unused by *encrypt* but it must be present.

SEE ALSO

login(1), passwd(1), getpass(3C) and passwd(4).

CAVEAT

The return value points to static data that are overwritten by each call.

NAME

ctermid – generate file name for terminal

SYNOPSIS

#include <stdio.h>

char *ctermid (s)
char *s;

DESCRIPTION

ctermid generates the path name of the controlling terminal for the current process, and stores it in a string.

If *s* is a NULL pointer, the string is stored in an internal static area, the contents of which are overwritten at the next call to *ctermid*, and the address of which is returned. Otherwise, *s* is assumed to point to a character array of at least **L_ctermid** elements; the path name is placed in this array and the value of *s* is returned. The constant **L_ctermid** is defined in the *<stdio.h>* header file.

NOTES

The difference between *ctermid* and *ttyname*(3C) is that *ttyname* must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while *ctermid* returns a string (**/dev/tty**) that will refer to the terminal if used as a file name. Thus *ttyname* is useful only if the process already has at least one file open to a terminal.

SEE ALSO

ttyname(3C).

This page is intentionally left blank

NAME

ctime, localtime, gmtime, asctime, tzset – convert date and time to string

SYNOPSIS

#include <time.h>

char *ctime (clock)
  time_t *clock;

struct tm *localtime (clock)
  time_t *clock;

struct tm *gmtime (clock)
  time_t *clock;

char *asctime (tm)
  struct tm *tm

extern time_t timezone, altzone;

extern int daylight;

extern char *tzname[ ];

void tzset ( );

DESCRIPTION

ctime, *localtime*, and *gmtime* accept arguments of type *time_t*, pointed to by *clock*, representing the time in seconds since 00:00:00 UTC, January 1, 1970. *ctime* returns a pointer to a 26-character string as shown below. Time zone and daylight savings corrections are made before the string is generated. The fields are constant in width:

```
Fri Sep 13 00:00:00 1986\n\0
```

localtime and *gmtime* return pointers to *tm* structures, described below. *localtime* corrects for the main time zone and possible alternate ("daylight saving") time zone; *gmtime* converts directly to Coordinated Universal Time (UTC), which is the time the UNIX system uses internally.

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

Declarations of all the functions and externals, and the *tm* structure, are in the <*time.h*> header file. The structure declaration is:

```
struct tm {
   int tm_sec;    /* seconds after the minute — [0, 61] */
                  */ for leap seconds */
   int tm_min;    /* minutes after the hour — [0, 59] */
   int tm_hour;   /* hour since midnight — [0, 23] */
   int tm_mday;   /* day of the month — [1, 31] */
   int tm_mon;    /* months since January — [0, 11] */
   int tm_year;   /* years since 1900 */
   int tm_wday;   /* days since Sunday — [0, 6] */
   int tm_yday;   /* days since January 1 — [0, 365] */
   int tm_isdst;  /* flag for alternate daylight */
                  */ saving time */
};
```

The value of *tm_isdst* is positive if daylight saving time is in effect, zero if daylight saving time is not in effect, and negative if the information is not available. (Previously, the value of *tm_isdst* was defined as non-zero if daylight saving time was in effect.)

The external *time_t* variable *altzone* contains the difference, in seconds, between Coordinated Universal Time and the alternate time zone. The external variable *timezone* contains the difference, in seconds, between UTC and local standard time. The external variable *daylight* indicates whether time should reflect daylight savings time. Both *timezone* and *altzone* default to 0 (UTC). The external variable *daylight* is non-zero if an alternate time zone exists. The time zone names are contained in the external variable *tzname*, which by default is set to:

```
char *tzname[2] = { "GMT", "   " };
```

These functions know about the peculiarities of this conversion for various time periods for the U.S.A (specifically, the years 1974, 1975, and 1987). They will handle the new daylight saving time starting with the first Sunday in April, 1987.

tzset uses the contents of the environment variable *TZ* to override the value of the different external variables. The function *tzset* is called by *asctime* and may also be called by the user. See *environ*(5) for a description of the *TZ* environment variable.

tzset scans the contents of the environment variable and assigns the different fields to the respective variable. For example, the most complete setting for New Jersey in 1986 could be

EST5EDT4,116/2:00:00,298/2:00:00

or simply

EST5EDT

An example of a southern hemisphere setting such as the Cook Islands could be

KDT9:30KST10:00,63/5:00,302/20:00

In the longer version of the New Jersey example of *TZ* *tzname*[0] is EST, *timezone* will be set to $5 * 60 * 60$, *tzname*[1] is EDT, *altzone* will be set to $4 * 60 * 60$, the starting date of the alternate time zone is the 117th day at 2 AM, the ending date of the alternate time zone is the 299th day at 2 AM (using zero-based Julian days), and *daylight* will be set positive. Starting and ending times are relative to the alternate time zone. If the alternate time zone start and end dates and the time are not provided, the days for the United States that year will be used and the time will be 2 AM. If the start and end dates are provided but the time is not provided, the time will be 2 AM. The effects of *tzset* are thus to change the values of the external variables *timezone*, *altzone*, *daylight* and *tzname*. *ctime*, *localtime*, *mktime* and *strftime* will also update these external variables as if they had called *tzset* at the time specified by the *time_t* or *struct tm* value that they are converting.

Note that on the Supermax, *TZ* is set to the correct value by default when the user logs on, via the local */etc/profile* file [see *profile*(4) and *timezone*(4)].

FILES

/usr/lib/locale/*language*/LC_TIME – file containing locale specific date and time information

SEE ALSO

time(2), getenv(3C), mktime(3C), putenv(3C), setlocale(3C), strftime(3C), printf(3S), cftime(4), profile(4), timezone(4), environ(5).

NOTES

The return values for *ctime*, *localtime* and *gmtime* point to static data whose content is overwritten by each call.

Setting the time during the interval of change from *timezone* to *altzone* or vice versa can produce unpredictable results. The system administrator must change the Julian start and end days annually.

NAME

ctype: isdigit, isxdigit, islower, isupper, isalpha, isalnum, isspace, iscntrl, ispunct, isprint, isgraph, isascii, setchrclass – character handling

SYNOPSIS

```
#include <ctype.h>

int isalpha (c)
  int c;

int isupper (c)
  int c;

int islower (c)
  int c;

int isdigit (c)
  int c;

int isxdigit (c)
  int c;

int isalnum (c)
  int c;

int isspace (c)
  int c;

int ispunct (c)
  int c;

int isprint (c)
  int c;

int isgraph (c)
  int c;

int iscntrl (c)
  int c;

int isascii (c)
  int c;

int setchrclass (chrclass)
char *chrclass;
```

DESCRIPTION

These macros classify character-coded integer values. Each is a predicate returning non-zero for true, zero for false. The behavior of these macros, except *isascii*, is affected by the current locale [see *setlocale*(3C)].

To modify the behavior, change the LC_TYPE category in *setlocale*(3C), that is, *setlocale* (LC_CTYPE, *newlocale*). In the *"C"* locale, or in a locale where character type information is not defined, characters are classified according to the rules of the US-ASCII 7-bit coded character set.

The macro *isascii* is defined on all integer values; the rest are defined only where the argument is an *int*, the value of which is representable as an *unsigned char*, or *EOF*, which is defined by the *stdio.h* header file and represents end-of-file.

isalnum tests for any character for which *isalpha* or *isdigit* is true (letter or digit).

isalpha tests for any character for which *isupper* or *islower* is true, or any character that is one of an implementation-defined set of characters for which none of *iscntrl*, *isdigit*, *ispunct*, or *isspace* is true. In the *"C"* locale, *isalpha* returns true only for the characters for which *isupper* or *islower* is true.

isascii tests for any ASCII character, code between *0* and *0177* inclusive.

iscntrl tests for any "control character" as defined by the character set.

isdigit tests for any decimal-digit character.

isgraph tests for any printing character, except space.

islower tests for any character that is a lower-case letter or is one of an implementation-defined set of characters for which none of *iscntrl*, *isdigit*, *ispunct*, *isspace*, or *isupper* is true. In the *"C"* locale, *islower* returns true only for the characters defined as lower-case ASCII characters.

isprint tests for any printing character, including space (" ").

ispunct tests for any printing character which is neither a space nor a character for which *isalnum* is true.

isspace tests for any space, tab, carriage-return, newline, vertical-tab or form-feed (standard white-space characters) or for one of an implementation-defined set of characters for which *isalnum* is false. In the *"C"* locale, *isspace* returns true only for the standard white-space characters.

isupper tests for any character that is an upper-case letter or is one of an implementation-defined set of characters for which none of *iscntrl*, *isdigit*, *ispunct*, *isspace*, or *islower* is true. In the *"C"* locale, *isupper* returns true only for the characters defined as upper-case ASCII characters.

isxdigit tests for any hexadecimal-digit character (**[0-9]**, **[A-F]** or **[a-f]**).

setchrclass initializes the table used by these functions and macros to a specific character classification set. *setchrclass* uses the value of its argument or the value of the environment variable **CHRCLASS** as the name of the datafile containing the information for the desired character set. Thes edatafiles are searched for in the special directory /lib/chrclass.

If *chrclass* is (char *)0, the value of the environment variable **CHRCLASS** is used. If **CHRCLASS** is not set **or** is undefined, the table retains its current value, which at initialization time is **iso.8859.1**, which describes the ISO 8859/1 character set (see *iso-8859/1*(5)).

All the character classification macros and the conversion functions and macros use a table lookup.

Functions exist for all the above defined macros. To get the function form, the macro name must be undefined (e.g., *#undef isdigit*).

FILES

/usr/lib/locale/*locale*/LC_CTYPE

/lib/chrclass – directory containing the datafiles for *setchrclass*

SEE ALSO

chrtbl(1M), setlocale(3C), stdio(3S), ascii(5), environ(5), iso-8859/1(5).

DIAGNOSTICS

If the argument to any of the character handling macros is not in the domain of the function, the result is undefined.

If *setchrclass* does not successfully fill the table, the table will not change (initially "iso.8859.1") and −1 is returned. If everything works, *setchrclass* returns 0.

NOTE

setchrclass is provided for compatibility with older versions and should not be used in new applications.

NAME

curses – terminal screen handling and optimization package

SYNOPSIS

The *curses* manual page is organized as follows:

In SYNOPSIS:

- compiling information
- summary of parameters used by *curses* routines
- alphabetical list of curses routines, showing their parameters

In DESCRIPTION:

- An overview of how *curses* routines should be used

In ROUTINES, descriptions of each *curses* routines, are grouped under the appropriate topics:

- Overall Screen Manipulation
- Window and Pad Manipulation
- Output
- Input
- Output Options Setting
- Input Options Setting
- Environment Queries
- Soft Labels
- Low-level Curses Access
- Terminfo-Level Manipulations
- Termcap Emulation
- Miscellaneous
- Use of **curscr**

Then come sections on:

- ATTRIBUTES
- COLORS
- FUNCTION KEYS
- LINE GRAPHICS

```
cc [flag ...] file ... -lcurses [library ...]
```

#include <curses.h> (automatically includes **<stdio.h>**, **<termio.h>**, and **<unctrl.h>**).

The parameters in the following list are not global variables, but rather this is a summary of the parameters used by the *curses* library routines. All routines return the **int** values **ERR** or **OK** unless otherwise noted. Routines that return pointers always return **NULL** on error. (**ERR**, **OK**, and **NULL** are all defined in **<curses.h>**)

bool bf
char * * area, * boolnames[ ], * boolcodes[ ], * boolfnames[ ],
  * bp
char * cap, * capname, codename[2], erasechar, * filename,
  * fmt
char * keyname, killchar, * label, * longname
char * name, * numnames[ ], * numcodes[ ], * numfnames[ ]
char * slk_label, * str, * strnames[ ], * strcodes[ ],
  * strfnames[ ]
char * term, * tgetstr, * tigetstr, * tgoto, * tparm, * type
chtype attrs, ch, horch, vertch
FILE * infd, * outfd
int begin_x, begin_y, begline, bot, c, col, count
int dmaxcol, dmaxrow, dmincol, dminrow, * errret, fildes
int (* init( )), labfmt, labnum, line
int ms, ncols, new, newcol, newrow, nlines, numlines
int oldcol, oldrow, overlay
int p1, p2, p9, pmincol, pminrow, (* putc( )), row
int smaxcol, smaxrow, smincol, sminrow, start
int tenths, top, visibility, x, y
short pair, f, b, color, r, g, b
SCREEN * new, * newterm, * set_term
TERMINAL * cur_term, * nterm, * oterm
va_list varglist

WINDOW * curscr, * dstwin, * initscr, * newpad, * newwin, * orig
WINDOW * pad, * srcwin, * stdscr, * subpad, * subwin, * win

addch(ch)
addstr(str)
attroff(attrs)
attron(attrs)
attrset(attrs)
baudrate()
beep()
box(win, vertch, horch)
can_change_color()
cbreak()
clear()
clearok(win, bf)
clrtobot()
clrtoeol()
color_content(color, &r, &g, &b)
copywin(srcwin, dstwin, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)"
curs_set(visibility)
def_prog_mode()
def_shell_mode()
del_curterm(oterm)
delay_output(ms)
delch()
deleteln()
delwin(win)
doupdate()
draino(ms)
echo()
echochar(ch)
endwin()
erase()
erasechar()
filter()

flash()
flushinp()
garbagedlines(win, begline, numlines)
getbegyx(win, y, x)
getch()
getmaxyx(win, y, x)
getstr(str)
getsyx(y, x)
getyx(win, y, x)
halfdelay(tenths)
has_colors()
has_ic()
has_il()
idlok(win, bf)
inch()
init_color(color, r, g, b)
init_pair(pair, f, b)
initscr()
insch(ch)
insertln()
intrflush(win, bf)
isendwin()
keyname(c)
keypad(win, bf)
killchar()
leaveok(win, bf)
longname()
meta(win, bf)
move(y, x)
mvaddch(y, x, ch)
mvaddstr(y, x, str)
mvcur(oldrow, oldcol, newrow, newcol)
mvdelch(y, x)
mvgetch(y, x)
mvgetstr(y, x, str)
mvinch(y, x)
mvinsch(y, x, ch)
mvprintw(y, x, fmt [, arg...])

mvscanw(y, x, fmt [, arg...])
mvwaddch(win, y, x, ch)
mvwaddstr(win, y, x, str)
mvwdelch(win, y, x)
mvwgetch(win, y, x)
mvwgetstr(win, y, x, str)
mvwin(win, y, x)
mvwinch(win, y, x)
mvwinsch(win, y, x, ch)
mvwprintw(win, y, x, fmt [, arg...])
mvwscanw(win, y, x, fmt [, arg...])
napms(ms)
newpad(nlines, ncols)
newterm(type, outfd, infd)
newwin(nlines, ncols, begin_y, begin_x)
nl()
nocbreak()
nodelay(win, bf)
noecho()
nonl()
noraw()
notimeout(win, bf)
overlay(srcwin, dstwin)
overwrite(srcwin, dstwin)
pair_content(pair, &f, &b)
pechochar(pad, ch)
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)"
prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)"
printw(fmt [, arg...])
putp(str)
raw()
refresh()
reset_prog_mode()
reset_shell_mode()
resetty()
restartterm(term, fildes, errret)

ripoffline(line, init)
savetty( )
scanw(fmt [, arg...])
scr_dump(filename)
scr_init(filename)
scr_restore(filename)
scroll(win)
scrollok(win, bf)
set_curterm(nterm)
set_term(new)
setscrreg(top, bot)
setsyx(y, x)
setupterm(term, fildes, errret)
slk_clear( )
slk_init(fmt)
slk_label(labnum)
slk_noutrefresh( )
slk_refresh( )
slk_restore( )
slk_set(labnum, label, fmt)
slk_touch( )
standend( )
standout( )
start_color( )
subpad(orig, nlines, ncols, begin_y, begin_x)
subwin(orig, nlines, ncols, begin_y, begin_x)
tgetent(bp, name)
tgetflag(codename)
tgetnum(codename)
tgetstr(codename, area)
tgoto(cap, col, row)
tigetflag(capname)
tigetnum(capname)
tigetstr(capname)
touchline(win, start, count)
touchwin(win)
tparm(str, p1, p2, ..., p9)
tputs(str, count, putc)

traceoff( )
traceon( )
typeahead(fildes)
unctrl(c)
ungetch(c)
vidattr(attrs)
vidputs(attrs, putc)
vwprintw(win, fmt, varglist)
vwscanw(win, fmt, varglist)
waddch(win, ch)
waddstr(win, str)
wattroff(win, attrs)
wattron(win, attrs)
wattrset(win, attrs)
wclear(win)
wclrtobot(win)
wclrtoeol(win)
wdelch(win)
wdeleteln(win)
wechochar(win, ch)
werase(win)
wgetch(win)
wgetstr(win, str)
winch(win)
winsch(win, ch)
winsertln(win)
wmove(win, y, x)
wnoutrefresh(win)
wprintw(win, fmt [, arg...])
wrefresh(win)
wscanw(win, fmt [, arg...])
wsetscrreg(win, top, bot)
wstandend(win)
wstandout(win)

DESCRIPTION

The *curses* routines give the user a terminal-independent method of updating screens with reasonable optimization.

In order to initialize the routines properly, **# include <curses.h>** must be included at the beginning of files that use any *curses* routines. In addition, the routine **initscr**( ) or **newterm**( ) must be called before any of the other routines that deal with windows and screens are used. (Three exceptions are noted where they apply.) The routine **endwin**( ) must be called before exiting. To get character-at-a-time input without echoing (most interactive, screen-oriented programs want this), after calling **initscr**( ) you should call "**cbreak**( ); **noecho**( );" Most programs would additionally call "**nonl**( ); **intrflush (stdscr, FALSE); keypad(stdscr, TRUE);**".

Before a *curses* program is run, a terminal's tab stops should be set and its initialization strings, if defined, must be output. This can be done by executing the **tput init** command after the shell environment variable **TERM** has been exported. For further details, see *profile*(4), *tput*(1), and the "Tabs and Initialization" subsection of *ter info*(4).

The *curses* library contains routines that manipulate data structures called *windows* that can be thought of as two-dimensional arrays of characters representing all or part of a terminal screen. A default window called **stdscr** is supplied, which is the size of the terminal screen. Others may be created with **newwin**( ). Windows are referred to by variables declared as **WINDOW** *; the type **WINDOW** is defined in **<curses.h >** to be a structure. These data structures are manipulated with routines described below, among which the most basic are **move**( ) and **addch**( ). (More general versions of these routines are included with names beginning with **w**, allowing you to specify a window. The routines not beginning with **w** usually affect **stdscr**). Then **refresh**( ) is called, telling the routines to make the user's terminal screen look like **stdscr**. The characters in a window are actually of type **chtype**, so that other information about the character may also be stored with each character.

Special windows called *pads* may also be manipulated. These are windows which are not constrained to the size of the screen and whose contents need not be displayed completely. See the description of **newpad**( ) under "Window and Pad Manipulation" for more information.

In addition to drawing characters on the screen, video attributes may be included which cause the characters to show up in modes such as underlined or in reverse video on terminals that support such display enhancements. Line drawing characters may be specified to be output. On input, *curses* is also able to translate arrow and function keys that transmit escape sequences into single values. The video attributes, line drawing characters, and input values use names, defined in **<curses.h>**, such as **A_REVERSE**, **ACS_HLINE**, and **KEY_LEFT**.

Routines that manipulate color on color alphanumeric terminals are new in this release of *curses*. to use these routines **start_color**( ) must be called, usually right after **initscr**( ). Colors are always used in pairs (referred to as color-pairs. A color-pairs consist of a foreground color (for characters) and a background color (for the field the characters are displayed on). A programmer initializes a color-pair with the routine **init_pair**( ). After it has been initialized, **COLOR_PAIR**(*n*), a macro defined in **<curses.h>**, can be used in the same ways other video attributes can be used. If a terminal is capable of redefining colors the programmer can use the routine **init_color**( ) to change the definition of a color. The routines **has_color**( ) and **can_change_color**( ) return **TRUE** or **FALSE**, depending on whether the terminal has color capabilities and whether the user can change the colors. The routine **color_content**( ) allows a user to identify the amounts of red, green, and blue components in an initialized color. The routine **pair_content**( ) allows a user to find out how a given color-pair is currently defined.

curses also defines the **WINDOW** * variable, **curscr**, which is used only for certain low-level operations like clearing and redrawing a garbaged screen. **curscr** can be used in only a few routines. If the window argument to **clearok**( ) is **curscr**, the next call to **wrefresh**( ) with any window will cause the screen to be cleared and repainted from scratch. If the window argument to **wrefresh**( ) is **curscr**, the screen in immediately cleared and repainted from scratch. This is how most programs would implement a "repaint-screen" function. More information on using **curscr** is provided where its use is appropriate.

The environment variables **LINES** and **COLUMNS** may be set to override **terminfo**'s idea of how large a screen is. These may be used in an AT&T Teletype 5620 layer, for example, where the size of a screen is changeable.

If the environment variable **TERMINFO** is defined, any program using *curses* will check for a local terminal definition before checking in the standard place.
For example, if the environment variable **TERM** is set to **att4425**, then the compiled terminal definition is found in */usr/lib/terminfo/a/att4425*. (The **a** is copied from the first letter of **att4425** to avoid creation of huge directories.) However, if **TERMINFO** is set to *$HOME/myterms*, *curses* will first check *HOME/myterms/a/att4425*, and, if that fails, will then check */usr/lib/terminfo/a/att4425*. This is useful for developing experimental definitions or when write permission on */usr/lib/terminfo* is not available.

The integer variables **LINES** and **COLS** are defined in **<curses.h>**, and will be filled in by **initscr**( ) with the size of the screen. (For more information, see the subsection "Terminfo-Level Manipulations".) The integer variables **COLORS** and **COLOR_PAIRS** are also defined in **<curses.h>** and contain, respectively, the maximum number of colors and color_pairs the terminal can support. They are initialized by **start_color**( ). The constants **TRUE** and **FALSE** have the values 1 and 0, respectively. The constants **ERR** and **OK** are returned by routines to indicate whether the routine successfully completed. These constants are also defined in **<curses.h>**.

ROUTINES

Many of the following routines have two or more versions. The routines prefixed with **w** require a *window* argument. The routines prefixed with **p** require a *pad* argument. Those without a prefix generally use **stdscr**.

The routines prefixed with **mv** require *y* and *x* coordinates to move to before performing the appropriate action. The **mv()** routines imply a call to **move()** before the call to the other routine. The window argument is always specified before the coordinates. *y* always refers to the row (of the window), and *x* always refers to the column. The upper left corner is always (**0**,**0**), not (**1**,**1**). The routines prefixed with **mvw** take both a *window* argument and *y* and *x* coordinates.

In each case, *win* is the window affected and *pad* is the pad affected. (**win** and **pad** are always of type **WINDOW ***). Option-setting routines require a boolean flag *bf* with the value **TRUE** or **FALSE**. (*bf* is always of type **bool**.) The types **WINDOW**, **bool**, and **chtype** are defined in **<curses.h>**. See the SYNOPSIS for a summary of what types all variables are.

All routines return either the integer **ERR** or the integer **OK**, unless otherwise noted. Routines that return pointers always return **NULL** on error.

Sometimes the description of a routine refers to a second routine. If the routine referred to is prefixed with a **w**, then you should assume that other versions of the second routine behave similarly. For example, the description of **initscr()** refers to **wrefresh()**. This implies that the same result will occur if **refresh()** is called.

Overall Screen Manipulation

WINDOW * initscr() The first routine called should almost always be **initscr()**. (The exceptions are **slk_init()**, **filter()**, and **ripoffline()**.) This will determine the terminal type and initialize all *curses* data structures. **initscr()** also arranges that the first call to **wrefresh()** will clear the screen. If errors occur, **initscr()** will write an appropriate error message to standard error and exit; otherwise, a pointer to **stdscr** is returned. If the program wants an indication of error conditions, **newterm()** should be used instead of **initscr()**. **initscr()** should only be called once per application.

endwin() A program should always call **endwin()** before exiting or escaping from *curses* mode temporarily, to do a shell escape or *system*(3S) call, for example. This routine will restore *tty*(7) modes, move the cursor to the lower left corner of the screen and reset the terminal into the proper non-visual mode. To resume after a temporary escape, call **wrefresh()** or **doupdate()**.

isendwin() Returns **TRUE** if **endwin()** has been called without any subsequent calls to **wrefresh()**.

SCREEN * newterm(type, outfd, infd)
A program that outputs to more than one terminal must use **newterm()** for each terminal instead of **initscr()**. A program that wants an indication of error conditions, so that it may

continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, must also use this routine. **newterm()** should be called once for each terminal. It returns a variable of type **SCREEN** * that should be saved as a reference to that terminal. The arguments are the *type* of the terminal to be used in place of the environment variable **TERM**; *outfd*, a *stdio*(3S) file pointer for output to the terminal; and *infd*, another file pointer for input from the terminal. When it is done running, the program must also call **endwin()** for each terminal being used. If **newterm()** is called more than once for the same terminal, the first terminal referred to must be the last one for which **endwin()** is called.

SCREEN * **set_term**(new)

This routine is used to switch between different terminals. The screen reference *new* becomes the new current terminal. A pointer to the screen of the previous terminal is returned by the routine. This is the only routine which manipulates SCREEN pointers; all other routines affect only the current terminal.

Window and Pad Manipulation

refresh()

wrefresh (win)

These routines (or **prefresh**(), **pnoutrefresh**(), **wnoutrefresh**(), or **doupdate**()) must be called to write output to the terminal, as most other routines merely manipulate data structures. **wrefresh**( ) copies the named window to the physical terminal screen, taking into account what is already there in order to minimize the amount of information that's sent to the terminal (called optimization). **refresh**( ) does the same thing, except it uses **stdscr** as a default window. Unless **leaveok**( ) has been enabled, the physical cursor of the terminal is left at the location. The number of characters output to the terminal is returned.

Note that **refresh**( ) is a macro.

wnoutrefresh(win)
doupdate( )

These two routines allow multiple updates to the physical terminal screen with more efficiency than **wrefresh**( ) alone. How this is accomplished is described in the next paragraph.

curses keeps two data structures representing the terminal screen: a *physical* terminal screen, describing what is actually on the screen, and a *virtual* terminal screen, describing what the programmer wants to have on the screen. **wrefresh**( ) works by first calling **wnoutrefresh**( ), which copys the named window to the virtual screen, and then by calling **doupdate**( ), which compares the virtual screen to the physical screen and does the actual update. If the programmer wishes to output several windows at once, a series of calls to **wrefresh**( ) will result in alternating calls to **wnoutrefresh**( ) and

doupdate( ), causing several bursts of output to the screen. By first calling **wnoutrefresh**( ) for each window, it is then possible to call **doupdate**( ) once, resulting in only one burst of output, with probably fewer total characters transmitted and certainly less processor time used.

WINDOW * **newwin**(nlines, ncols, begin_y, begin_x)
Create and return a pointer to a new window with the given number of lines (or rows), *nlines*, and columns, *ncols*. The upper left corner of the window is at line *begin_y*, column *begin_x*. If either *nlines* or *ncols* is **0**, they will be set to the value of **lines**−*begin_y* and **cols**−*begin_x*. A new full-screen window is created by calling **newwin(0,0,0,0)**.

mvwin(win, y, x)
Move the window so that the upper left corner will be at position (*y*, *x*). If the move would cause any portion of the window to be moved off the screen, it is an error and the window is not moved.

WINDOW * **subwin**(orig, nlines, ncols, begin_y, begin_x)
Create and return a pointer to a new window with the given number of lines (or rows), *nlines*, and columns, *ncols*. The window is at position (*begin_y*, *begin_x*) on the screen. (This position is relative to the screen, and not to the window *orig*.) The window is made in the middle of the window *orig*, so that changes made to one window will affect the character image of both windows. When changing the image of a subwindow, it will be necessary to call

touchwin() or **touchline**() on *orig* before calling **wrefresh**() on *orig*.

delwin(win)

Delete the named window, freeing up all memory associated with it. If you try to delete a main window before all of its subwindows have been deleted, ERR will be returned.

WINDOW * **newpad**(nlines, ncols)

Create and return a pointer to a new pad data structure with the given number of lines (or rows), *nlines*, and columns, *ncols*. A pad is a window that is not restricted by the screen size and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (e.g. from scrolling or echoing of input) do not occur. It is not legal to call **wrefresh**() with a pad as an argument; the routines **prefresh**() or **pnoutrefresh**() should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

WINDOW * **subpad**(orig, nlines, ncols, begin_y, begin_x)

Create and return a pointer to a subwindow within a pad with the given number of lines (or rows), *nlines*, and columns, *ncols*. Unlike **subwin**(), which uses screen coordinates, the window is at position (*begin_y, begin_x*) on the pad. The window is made in the middle of the window *orig*, so that changes made to one window will affect the character image of both windows. When changing the image of a subwindow, it will be necessary to call **touchwin**() or **touchline**() on *orig* before calling **prefresh**() on *orig*.

prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)

pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)

These routines are analogous to **wrefresh**() and **wnoutrefresh**() except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left corner, in the pad, of the rectangle to be displayed. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges, on the screen, of the rectangle to be displayed in. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

Output

These routines are used to manipulate text in windows.

addch(ch)
waddch(win, ch)
mvaddch(y, x, ch)
mvwaddch(win, y, x, ch)

The character *ch* is put into the window at the current cursor position of the window and the position of the window cursor is advanced. Its function is similar to that of *putchar* (see *putc*(3S)). At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok**( ) is enabled, the scrolling region will be scrolled up one line.

If *ch* is a tab, newline, or backspace, the cursor will be moved appropriately within the window. A newline also does a **wclrtoeol**( ) before moving. Tabs are considered to be at every eighth column. If *ch* is another control character, it will be drawn in the ^X notation. (Calling **winch**( ) on a position in the window containing a control character will not return the control character, but instead will return one character of the representation of the control character.)

Video attributes can be combined with a character by or-ing them into the parameter. This will result in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another using **winch**( ) and **waddch**( ).) See

wstandout( ).

Note that *ch* is actually of type **chtype**, not a character.

Note that **addch**( ), **mvaddch**( ), and **mvwaddch**( ), are macros.

echochar(ch)
wechochar(win, ch)
pechochar(pad, ch)

These routines are functionally equivalent to a call to **addch**(ch) followed by a call to **refresh**( ), a call to **waddch**(win, ch) followed by a call to **wrefresh**(win), or a call to **waddch**(pad, ch) followed by a call to **prefresh**(pad). The knowledge that only a single character is being output is taken into consideration and a considerable performance gain can be seen by using these routines instead of their equivalents. In the case of **pechochar**( ), the last location of the pad on the screen is reused for the arguments to **prefresh**( ).

Note that *ch* is actually of type **chtype**, not a character.

Note that **echochar**( ) is a macro.

addstr(str)
waddstr(win, str)
mvwaddstr(win, y, x, str)
mvaddstr(y, x, str)

These routines write all the characters of the null-terminated character string *str* on the given window. This is equivalent to calling **waddch**( ) once for each character in the string.

Note that **addstr**( ), **mvaddstr**( ), and **mvwaddstr**( ) are macros.

attroff(attrs)
wattroff(win, attrs)
attron(attrs)
wattron(win, attrs)
attrset(attrs)
wattrset(win, attrs)
standend()
wstandend(win)
standout()
wstandout(win)

These routines manipulate the current attributes of the named window. These attributes can be any combination of **A_STANDOUT**, **A_REVERSE**, **A_BOLD**, **A_DIM**, **A_BLINK**, **A_UNDERLINE**, and **A_ALTCHARSET**, as well as the macro **COLOR_PAIR**(n). These attributes are defined in < **curses.h** > and can be combined with the C logical OR (|) operator.

The current attributes of a window are applied to all characters that are written into the window with **waddch**(). Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they will be displayed as the graphic rendition of the characters put on the screen.

wattrset(win, attrs) sets the current attributes of the given window to *attrs*. **wattroff**(win, attrs) turns off the named attributes without turning on or off any other attributes. **wattron**(win, attrs) turns on the named attributes without affecting any others.

wstandout(win, attrs) is the same as **wattron**(win, A_STANDOUT).
wstandend(win, attrs) is the same as **wattrset**(win, **0**), that is, it turns off all attributes.

Note that **wattroff**( ), **wattron**( ), **wattrset**( ), **wstandend**( ), and **wstandout**( ) return **1** at all times.

Note that *attrs* is actually of type **chtype**, not a character.

Note that **attroff**( ), **attron**( ), **attrset**( ), **standend**( ), and **standout**( ) are macros.

beep( )
flash( )

These routines are used to signal the terminal user. **beep**( ) will sound the audible alarm on the terminal, if possible, and if not, will flash the screen (visible bell), if that is possible. **flash**( ) will flash the screen, and if that is not possible, will sound the audible signal. If neither signal is possible, nothing will happen. Nearly all terminals have an audible signal (bell or beep) but only some can flash the screen.

box(win, vertch, horch)

A box is drawn around the edge of the window, *win*. *vertch* and *horch* are the characters the box is to be drawn with. If *vertch* and *horch* are **0**, then appropriate default characters, **ACS_VLINE** and **ACS_HLINE**, will be used.

Note that *vertch* and *horch* are actually of type **chtype**, not characters.

erase( )
werase(win)

These routines copy blanks to every position in the window.

Note that **erase**( ) is a macro.

clear( )
wclear(win)

These routines are like **erase**( ) and **werase**( ), but they also call **clearok**( ), arranging that the screen will be cleared completely on the next call to **wrefresh**( ) for that window, and repainted from scratch.

Note that **clear**( ) is a macro.

clrtobot( )
wclrtobot(win)

All lines below the cursor in this window are erased. Also, the current line to the right of the cursor, inclusive, is erased.

Note that **clrtobot**( ) is a macro.

clrtoeol( )
wclrtoeol(win)

The current line to the right of the cursor, inclusive, is erased.

Note that **clrtoeol**( ) is a macro.

delay_output(ms)

Insert a *ms* millisecond pause in the output. It is not recommended that this routine be used extensively, because padding characters are used rather than a processor pause.

delch( )
wdelch(win)
mvdelch(y, x)
mvwdelch(win, y, x)

The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change (after moving to (*y*, *x*), if specified). (This does not imply use of the hardware "delete-character" feature.)

Note that **delch**( ), **mvdelch**( ), and **mvwdelch**( ) are macros.

deleteln( )
wdeleteln(win)

The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change. (This does not imply use of the hardware "delete-line" feature.)

Note that **deleteln**( ) is a macro.

getyx(win, y, x)

The cursor position of the window is placed in the two integer variables *y* and *x*.

Note that **getyx**( ) is a macro, so no "**&**" is necessary before the variables *y* and *x*.

getbegyx(win, y, x)
getmaxyx(win, y, x)

The current beginning coordinates (**getbegyx**( )) or size (**getmaxyx**( )) of the specified window are placed in the two integer variables *y* and *x*.

Note that **getbegyx**( ) and **getmaxyx**( ) are macros, so no "**&**" is necessary before the variables *y* and *x*.

insch(ch)
winsch(win, ch)
mvwinsch(win, y, x, ch)
mvinsch(y, x, ch)

The character *ch* is inserted before the character under the cursor. All characters to the right are moved one space to the right, losing the rightmost character of the line. The cursor position does not change (after moving to (*y*, *x*), if specified). (This does not imply use of the hardware "insert-character" feature.)

Note that *ch* is actually of type **chtype**, not a character.

Note that **insch**( ), **mvinsch**( ), and **mvwinsch**( ) are macros.

insertln( )
winsertln(win)

A blank line is inserted above the current line and the bottom line is lost. (This does not imply use of the hardware "insert-line" feature.)

Note that **insertln**( ) is a macro.

move(y, x)
wmove(win, y, x)

The cursor associated with the window is moved to line (row) *y*, column *x*. This does not move the physical cursor of the terminal until **wrefresh**( ) is called. The position specified is relative to the upper left corner of the window, which is **(0, 0)**.

Note that **move**( ) is a macro.

overlay(srcwin, dstwin)
overwrite(srcwin, dstwin)

These routines overlay text from *srcwin* on top of text from *dstwin* wherever the two windows overlap. The difference is that **overlay**( ) is non-destructive (blanks are not copied), while **overwrite**( ) is destructive.

copywin(srcwin, dstwin, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)

This routine provides finer control over the **overlay**( ) and **overwrite**( ) routines. As in the **prefresh**( ) routine, a rectangle is specified in the destination window, (*dminrow*, *dmincol*) and (*dmaxrow*, *dmaxcol*), and the upper-left-corner coordinates of the source window, (*sminrow*, *smincol*). If the argument *overlay* is true, then copying is non-destructive, as in **overlay**( ).

printw(fmt [, arg...])
wprintw(win, fmt [, arg...])
mvprintw(y, x, fmt [, arg...])
mvwprintw(win, y, x, fmt [, arg...])

These routines are analogous to **printf**(3). The string which would be output by **printf**(3) is instead output using **waddstr**( ) on the given window.

vwprintw(win, fmt, varglist)

This routine corresponds to *vfprintf*(3S). It performs a **wprintw**( ) using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in **<varargs.h>**. See the *vprintf*(3S) and *varargs*(5) manual pages for a detailed description on how to use variable argument lists.

scroll(win)
: The window is scrolled up one line. This involves moving the lines in the window data structure.

touchwin(win)
touchline(win, start, count)
: Throw away all optimization information about which parts of the window have been touched, by pretending that the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window will affect the other window, but the records of which lines have been changed in the other window will not reflect the change. **touchline**( ) only pretends that *count* lines have been changed, beginning with line *start* .

Input

getch( )
wgetch(win)
mvgetch(y, x)
mvwgetch(win, y, x)
: A character is read from the terminal associated with the window. In NODELAY mode, if there is no input waiting, the value **ERR** is returned. In DELAY mode, the program will hang until the system passes text through to the program. Depending on the setting of **cbreak**( ), this will be after one character (CBREAK mode), or after the first newline (NOCBREAK mode). In HALF-DELAY mode, the program will hang until a character is typed or the

specified timeout has been reached. Unless **noecho**( ) has been set, the character will also be echoed into the designated window.

When **wgetch**( ) is called, before getting a character, it will call **wrefresh**( ) if anything in the window has changed (for example, the cursor has moved or text changed).

When using **getch**( ), **wgetch**( ), **mvgetch**( ), or **mvwgetch**( ), do not set both NOCBREAK mode (**nocbreak**( )) and ECHO mode (**echo**( )) at the same time. Depending on the state of the *tty*(7) driver when each character is typed, the program may produce undesirable results.

If **wgetch**( ) encounters a **^D**, it is returned (unlike *stdio* routines, which would return a null string and have a return code of **-1**).

If **keypad**(win, **TRUE**) has been called, and a function key is pressed, the token for that function key will be returned instead of the raw characters. (See **keypad**( ) under "Input Options Setting.") Possible function keys are defined in **<curses.h>** with integers beginning with **0401**, whose names begin with **KEY_**. If a character is received that could be the beginning of a function key (such as escape), *curses* will set a timer. If the remainder of the sequence is not received within the designated time, the character will be passed through, otherwise the function key value will be returned. For this reason, on many terminals, there will be a delay after a user presses the escape key before the escape is returned to the program. (Use by a programmer of the escape key for a single character routine is discouraged. Also see **notimeout**() below.)

Note that **getch**(), **mvgetch**(), and **mvwgetch**() are macros.

getstr(str)
wgetstr(win, str)
mvgetstr(y, x, str)
mvwgetstr(win, y, x, str)

A series of calls to **wgetch**() is made, until a newline, carriage return, or enter key is received. The resulting value (except for this terminating character) is placed in the area pointed at by the character pointer *str*. The user's erase and kill characters are interpreted. See **wgetch**() for how it handles characters differently from *stdio* routines (especially **^D**).

Note that **getstr**(), **mvgetstr**(), and **mvwgetstr**() are macros.

ungetch(c)

Place *c* onto the input queue, to be returned by the next call to **wgetch**().

flushinp()

Throws away any typeahead that has been typed by the user and has not yet been read by the program. Note that **flushinp**() will not throw away any characters supplied by **ungetch**().

inch( )
winch(win)
mvinch(y, x)
mvwinch(win, y, x)

The character, of type **chtype**, at the current position in the named window is returned. If any attributes are set for that position, their values will be OR'ed into the value returned. The predefined constants **A_CHARTEXT** and **A_ATTRIBUTES**, defined in < **curses.h** >, can be used with the C logical AND (&) operator to extract the character or attributes alone.

Note that **inch**( ), **winch**( ), **mvinch**( ), and **mvwinch**( ) are macros.

scanw(fmt [, arg...])
wscanw(win, fmt [, arg...])
mvscanw(y, x, fmt [, arg...])
mvwscanw(win, y, x, fmt [, arg...])

These routines correspond to *scanf*(3S), as do their arguments and return values. **wgetstr**( ) is called on the window, and the resulting line is used as input for the scan. The return value for these routines is the number of *arg* values that are converted by *fmt*. *arg* values that are not converted are lost. See **wgetstr**( ) for how it handles strings differently than the *stdio* routines (especially ^D).

vwscanw(win, fmt, ap)

This routine is similar to **vwprintw**( ) in that it performs a **wscanw**( ) using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in

<**varargs.h**>. See the *vprintf*(3S) and *varargs*(5) manual pages for a detailed description on how to use variable argument lists.

Output Options Setting

These routines set options within *curses* that deal with output. All options are initially FALSE, unless otherwise stated. It is not necessary to turn these options off before calling **endwin**().

clearok(win, bf)
: If enabled (*bf* is **TRUE**), the next call to **wrefresh**() with this window will clear the screen completely and redraw the entire screen from scratch. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

idlok(win, bf)
: If enabled (*bf* is **TRUE**), *curses* will consider using the hardware "insert/delete-line" feature of terminals so equipped. If disabled (*bf* is FALSE), *curses* will very seldom use this feature. (The "insert/delete-character" feature is always considered.) This option should be enabled only if your application needs "insert/delete-line", for example, for a screen editor. It is disabled by default because "insert/delete-line" tends to be visually annoying when used in applications where it isn't really needed. If "insert/delete-line" cannot be used, *curses* will redraw the changed portions of all lines. Not calling **idlok**() saves approximately 5000 bytes of memory.

leaveok(win, bf)
: Normally, the hardware cursor is left at the location of the window cursor being refreshed. This option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

setscrreg(top, bot)
wsetscrreg(win, top, bot)
: These routines allow the user to set a software scrolling region in a window. *top* and *bot* are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and **scrollok**( ) are enabled, an attempt to move off the bottom margin line will cause all lines in the scrolling region to scroll up one line. (Note that this has nothing to do with use of a physical scrolling region capability in the terminal, like that in the DEC VT100. Only the text of the window is scrolled; if **idlok**( ) is enabled and the terminal has either a scrolling region or "insert/delete-line" capability, they will probably be used by the output routines.)

  Note that **setscrreg**( ) is a macro.

scrollok(win, bf)
: This option controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either from a newline on the bottom line, or typing the last character of the last line. If disabled (*bf* is

FALSE), the cursor is left on the bottom line at the location where the offending character was entered. If enabled (*bf* is **TRUE**), **wrefresh**() is called on the window, and then the physical terminal and window are scrolled up one line. (Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call **idlok**( ).)

Note that **scrollok**() will always return **OK**.

Input Options Setting

These routines set options within *curses* that deal with input. The options involve using *ioctl*(2) and therefore interact with *curses* routines. It is not necessary to turn these options off before calling **endwin**( ).

For more information on these options, see the chapter of the *Programmer's Guide* that describes how to write *curses* programs.

cbreak()
nocbreak()

These two routines put the terminal into and out of CBREAK mode, respectively. In CBREAK mode, characters typed by the user are immediately available to the program and erase/kill character processing is not performed. When in NOCBREAK mode, the tty driver will buffer characters typed until a newline or carriage return is typed. Interrupt and flow-control characters are unaffected by this mode (see *termio*(7)). Initially the terminal may or may not be in CBREAK mode, as it is inherited, therefore, a program should call **cbreak** or **nocbreak** explicitly.

Most interactive programs using *curses* will set CBREAK mode.

Note that **cbreak**( ) performs a subset of the functionality of **raw**( ). See **wgetch**( ) under "Input" for a discussion of how these routines interact with **echo**( ) and **noecho**( ).

echo( )
noecho( )

These routines control whether characters typed by the user are echoed by **wgetch**( ) as they are typed. Echoing by the tty driver is always disabled, but initially **wgetch**( ) is in ECHO mode, so characters typed are echoed. Authors of most interactive programs prefer to do their own echoing in a controlled area of the screen, or not to echo at all, so they disable echoing by calling **noecho**( ). See **wgetch**( ) under "Input" for a discussion of how these routines interact with **cbreak**( ) and **nocbreak**( ).

nl( )
nonl( )

These routines control whether carriage return is translated into newline on input by **wgetch**( ). Initially, this translation is done; **nonl**( ) turns the translation off. Note that translation by the *tty*(7) driver is disabled in CBREAK mode.

halfdelay(tenths)

Half-delay mode is similar to CBREAK mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, ERR will be returned if nothing has been typed.

tenths must be a number between 1 and 255. Use **nocbreak**( ) to leave half-delay mode.

intrflush(win, bf) If this option is enabled, when an interrupt key is pressed on the keyboard (interrupt, break, quit) all output in the tty driver queue will be flushed, giving the effect of faster response to the interrupt, but causing *curses* to have the wrong idea of what is on the screen. Disabling the option prevents the flush. The default for the option is inherited from the tty driver settings. The window argument is ignored.

keypad(win, bf) This option enables *curses* to obtain information from the keypad of the user's terminal. If enabled, the user can press a function key (such as an arrow key) and **wgetch**() will return a single value representing the function key, as in **KEY_LEFT**. If disabled, *curses* will not treat function keys specially and the program would have to interpret the escape sequences itself. If the keypad in the terminal can be turned on (made to transmit), calling **keypad** (*win*, **TRUE**) will turn it on.

meta(win, bf) Initially, whether the terminal returns 7 or 8 significant bits on input depends on the control mode of the tty driver (see *termio*(7)). To force 8 bits to be returned, invoke **meta** (*win*, **TRUE**). To force 7 bits to be returned, invoke **meta** (*win*, **FALSE**). The window argument, *win*, is always ignored. If the *terminfo*(4) capabilities **smm** (meta_on) and **rmm** (meta_off) are defined for the terminal, **smm** will be sent to the terminal when **meta** (*win*, **TRUE**) is called and **rmm** will be sent when **meta** (*win*, **FALSE**) is called.

nodelay(win, bf)
This option causes **wgetch**() to be a non-blocking call. If no input is ready, **wgetch**() will return **ERR**. If disabled, **wgetch**() will hang until a key is pressed.

notimeout(win, bf)
While interpreting an input escape sequence, **wgetch**() will set a timer while waiting for the next character. If **notimeout**(*win*, **TRUE**) is called, then **wgetch**() will not set a timer. The purpose of the timeout is to differentiate between sequences received from a function key and those typed by a user.

raw()
noraw()
The terminal is placed into or out of RAW mode. RAW mode is similar to CBREAK mode, in that characters typed are passed through to the user program; however, in RAW mode, the interrupt, quit, suspend, and flow control characters are passed through uninterpreted, instead of generating a signal as they do in CBREAK mode. The behavior of the BREAK key depends on other bits in the *tty*(7) driver that are not set by *curses*.

typeahead(fildes)
curses does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update will be

postponed until **wrefresh**( ) or **doupdate**( ) is called again. This allows faster response to commands typed in advance. Normally, the file descriptor for the input FILE pointer passed to **newterm**( ), or **stdin** in the case that **initscr**( ) was used, will be used to do this typeahead checking. The **typeahead**( ) routine specifies that the file descriptor *fildes* is to be used to check for typeahead instead. If *fildes* is **−1**, then no typeahead checking will be done.

Note that *fildes* is a file descriptor, not a **<stdio.h>** FILE pointer.

Environment Queries

baudrate( )
Returns the output speed of the terminal. The number returned is in bits per second, for example, 9600, and is an integer.

char erasechar( )
The user's current erase character is returned.

has_ic( )
True if the terminal has insert- and delete-character capabilities.

has_il( )
True if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This might be used to check to see if it would be appropriate to turn on physical scrolling using **scrollok**( ) or **idlok**( ).

char killchar( )
The user's current line-kill character is returned.

char * longname() This routine returns a pointer to a static area containing a verbose description of the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to **initscr()** or **newterm()**. The area is overwritten by each call to **newterm()** and is not restored by **set_term()**, so the value should be saved between calls to **newterm()** if **longname()** is going to be used with multiple terminals.

Color Manipulation

This section describes the color manipulation routines introduced in this release of *curses*.

can_change_color() This routine requires no arguments. It returns **TRUE** if the terminal supports colors and can change their definitions, **FALSE** otherwise. This routine facilitates writing terminal-independent programs.

color_content(color, &r, &g, &b)

This routine gives users a way to find the intensity of the red, green and blue (RGB) components in a color. It requires four arguments: the color number, and three addresses of **shorts** for storing the information about the amounts of red, green, and blue components in the given color. The value of the first argument must be between 0 and **COLORS** – 1. The values that will be stored at the addresses pointed to by the last three arguments will be between 0 (no component) and 1000 (maximum amount of component). This routine returns **ERR** if the color does not exist (the first argument is outside the valid range), or if the terminal cannot change color definitions, **OK** otherwise.

has_color() This routine requires no arguments. It returns **TRUE** if the terminal can manipulate colors, **FALSE** otherwise. This routine facilitates writing terminal-independent programs. For example, a programmer can use it to decide whether to use color or some other video attribute.

init_color(color, r, g, b)
This routine changes the definition of a color. It takes four arguments: the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). (See section COLOR for the default color index). The value of the first argument must be between 0 and **COLORS** − 1. The last three arguments must each be a value between 0 and 1000. When **init_color**() is used, all occurrences of that color on the screen immediately change to the new definition. It returns **OK** if it was able to change the definition of the color, **ERR** otherwise.

init_pair(pair, f, b) This routine changes the definition of a color_pair. It takes three arguments: the number of the color_pair to be changed, the foreground color number, and the background color number. The value of the first argument must be between 1 and **COLOR_PAIRS** − 1. The value of the second and third arguments must be between 0 and **COLORS** – 1. If the color_pair was previously initialized, the screen will be refreshed and all occurrences of that color_pair will be changed to the new definition. The routine returns **OK** if it was able to change the definition of the color_pair, **ERR** otherwise.

pair_content(pair, &f, &b)
This routine allows users to find out what colors a given color_pair consists of. It requires three arguments: the color_pair number, and two addresses of **shorts** for storing the foreground and the background color numbers. The value of the first argument must be between 1 and **COLOR_PAIRS** – **1**. The values that will be stored at the addresses pointed to by the second and third arguments will be between 0 and **COLORS** – 1. The routine returns **ERR** if the color_pair has not been initialized, **OK** otherwise.

start_color()
This routine requires no arguments. It must be called if the user wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after **initscr**(). **start_color**() initializes eight basic colors (black, blue, green, cyan, red, magenta, yellow, and white), and two global variables. **COLORS** and **COLOR_PAIRS** (respectively defining the maximum number of colors and color_pairs the terminal can support). It also restores the terminal's colors to the values they had when the terminal

was just turned on. It returns **ERR** if the terminal does not support colors, **OK** otherwise.

Soft Labels

If desired, *curses* will manipulate the set of soft function-key labels that exist on many terminals. For those terminals that do not have soft labels, if you want to simulate them, *curses* will take over the bottom line of **stdscr**, reducing the size of **stdscr** and the variable **LINES**. *curses* standardizes on 8 labels of 8 characters each. If a *curses* program changes the values of the soft labels, it can restore them only to the default settings for that terminal. Therefore, if before calling a *curses* program a user changes the values of the soft labels, those values cannot be reset when the *curses* program terminates.

slk_init(labfmt) In order to use soft labels, this routine must be called before **initscr**() or **newterm**() is called. If **initscr**() winds up using a line from **stdscr** to emulate the soft labels, then *labfmt* determines how the labels are arranged on the screen. Setting *labfmt* to **0** indicates that the labels are to be arranged in a 3-2-3 arrangement; **1** asks for a 4-4 arrangement.

slk_set(labnum, label, labfmt)
labnum is the label number, from 1 to 8. *label* is the string to be put on the label, up to 8 characters in length. A **NULL** string or a **NULL** pointer will put up a blank label. *labfmt* is one of **0**, **1** or **2**, to indicate whether the label is to be left-justified, centered, or right-justified within the label.

slk_refresh( )
slk_noutrefresh( ) These routines correspond to the routines **wrefresh**( ) and **wnoutrefresh**( ). Most applications would use **slk_noutrefresh**( ) because a **wrefresh**( ) will most likely soon follow.

char * slk_label(labnum)
The current label for label number *labnum* is returned, in the same format as it was in when it was passed to **slk_set**( ); that is, how it looked prior to being justified according to the *labfmt* argument of **slk_set**( ).

slk_clear( ) The soft labels are cleared from the screen.

slk_restore( ) The soft labels are restored to the screen after a **slk_clear**( ).

slk_touch( ) All of the soft labels are forced to be output the next time a **slk_noutrefresh**( ) is performed.

Low-Level *curses* Access

The following routines give low-level access to various *curses* functionality. These routines typically would be used inside of library routines.

def_prog_mode( )
def_shell_mode( ) Save the current terminal modes as the "program" (in **curses**) or "shell" (not in **curses**) state for use by the **reset_prog_mode**( ) and **reset_shell_mode**( ) routines. This is done automatically by **initscr**( ).

reset_prog_mode()
reset_shell_mode() Restore the terminal to "program" (in **curses**) or "shell" (out of *curses*) state. These are done automatically by **endwin()** and **doupdate()** after an **endwin()**, so they normally would not be called.

resetty()
savetty() These routines save and restore the state of the terminal modes. **savetty()** saves the current state of the terminal in a buffer and **resetty()** restores the state to what it was at the last call to **savetty()**.

getsyx(y, x) The current coordinates of the virtual screen cursor are returned in *y* and *x*. If **leaveok()** is currently **TRUE**, then **−1**, **−1** will be returned. If lines have been removed from the top of the screen using **ripoffline()**, *y* and *x* include these lines; therefore, *y* and *x* should be used only as arguments for **setsyx()**.

Note that **getsyx()** is a macro, so no "**&**" is necessary before the variables *y* and *x*.

setsyx(y, x) The virtual screen cursor is set to *y*, *x*. If *y* and *x* are both **−1**, then **leaveok()** will be set. The two routines **getsyx()** and **setsyx()** are designed to be used by a library routine which manipulates *curses* windows but does not want to change the current position of the program's cursor. The library routine would call **getsyx()** at the beginning, do its manipulation of its own windows, do a **wnoutrefresh**() on its windows, call **setsyx**(), and then call **doupdate**().

ripoffline(line, init) This routine provides access to the same facility that **slk_init**() uses to reduce the size of the screen. **ripoffline**() must be called before **initscr**() or **newterm**() is called. If *line* is positive, a line will be removed from the top of **stdscr**; if negative, a line will be removed from the bottom. When this is done inside **initscr**(), the routine *init*() is called with two arguments: a window pointer to the 1-line window that has been allocated and an integer with the number of columns in the window. Inside this initialization routine, the integer variables **LINES** and **COLS** (defined in **<curses.h>**) are not guaranteed to be accurate and **wrefresh**() or **doupdate**() must not be called. It is allowable to call **wnoutrefresh**() during the initialization routine.

ripoffline() can be called up to five times before calling **initscr**() or **newterm**().

scr_dump(filename) The current contents of the virtual screen are written to the file *filename*.

scr_restore(filename) The virtual screen is set to the contents of *filename*, which must have been written using **scr_dump**(). **ERR** is returned if the contents of *filename* are not compatible with the current release of *curses* software. The next call to **doupdate**() will restore the screen to what it looked like in the dump file.

scr_init(filename) The contents of *filename* are read in and used to initialize the *curses* data structures about what the terminal currently has on its screen. If the data is determined to be valid, *curses* will base its next update of the screen on this information rather than clearing the screen and starting from scratch. **scr_init**() would be used after **initscr**() or a *system*(3S) call to share the screen with another process which has done a **scr_dump**() after its **endwin**() call. The data will be declared invalid if the *terminfo*(4) capability **nrrmc** is true or the time-stamp of the tty is old. Note that **keypad**(), **meta**(), **slk_clear**(), **curs_set**(), **flash**(), and **beep**() do not affect the contents of the screen, but will make the tty's time-stamp old.

curs_set(visibility) The cursor state is set to invisible, normal, or very visible for *visibility* equal to **0**, **1** or **2**. If the terminal supports the *visibility* requested, the previous *cursor* state is returned; otherwise, **ERR** is returned.

draino(ms) Wait until the output has drained enough that it will only take *ms* more milliseconds to drain completely.

garbagedlines(win, begline, numlines)
This routine indicates to *curses* that a screen line is garbaged and should be thrown away before written over the top of it. It could be used for programs such as editors which want a command

to redraw just a single line. Such a command could be used in cases where there is a noisy communications line and redrawing the entire screen would be subject to even more communication noise. Just redrawing the single line gives some semblance of hope that it would show up unblemished. The current location of the window is used to determine which lines are to be redrawn.

napms(ms) Sleep for *ms* milliseconds.

mvcur(oldrow, oldcol, newrow, newcol)
Low-level cursor motion.

Terminfo-Level Manipulations

These low-level routines must be called by programs that need to deal directly with the *terminfo*(4) database to handle certain terminal capabilities, such as programming function keys. For all other functionality, *curses* routines are more suitable and their use is recommended.

Initially, **setupterm**( ) should be called. (Note that **setupterm**( ) is automatically called by **initscr**( ) and **newterm**( ).) This will define the set of terminal-dependent variables defined in the *terminfo*(4) database. The *terminfo*(4) variables **lines** and **columns** (see *terminfo*(4)) are initialized by **setupterm**( ) as follows: if the environment variables **LINES** and **COLUMNS** exist, their values are used. If the above environment variables do not exist and the program is running in a layer (see *layers*(1)), the size of the current layer is used. Otherwise, the values for **lines** and **columns** specified in the *terminfo*(4) database are used.

The header files <**curses.h**> and <**term.h**> should be included, in this order, to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through **tparm**( ) to instantiate them. All *terminfo*(4) strings (including the output of **tparm**( )) should be printed with

tputs( ) or **putp**( ). Before exiting, **reset_shell_mode**( ) should be called to restore the tty modes. Programs which use cursor addressing should output **enter_ca_mode** upon startup and should output **exit_ca_mode** before exiting (see *terminfo*(4)). (Programs desiring shell escapes should call **reset_shell_mode**( ) and output **exit_ca_mode** before the shell is called and should output **enter_ca_mode** and call **reset_prog_mode**( ) after returning from the shell. Note that this is different from the *curses* routines (see **endwin**( )).

setupterm(term, fildes, errret)

Reads in the *terminfo*(4) database, initializing the *terminfo*(4) structures, but does not set up the output virtualization structures used by *curses*. The terminal type is in the character string *term*; if *term* is **NULL**, the environment variable **TERM** will be used. All output is to the file descriptor *fildes*. If *errret* is not **NULL**, then **setupterm**( ) will return **OK** or **ERR** and store a status value in the integer pointed to by *errret*. A status of **1** in *errret* is normal, **0** means that the terminal could not be found, and **−1** means that the *terminfo*(4) database could not be found. If *errret* is **NULL**, **setupterm**( ) will print an error message upon finding an error and exit. Thus, the simplest call is **setupterm ((char *)0, 1, (int *)0)**, which uses all the defaults.

The *terminfo*(4) boolean, numeric and string variables are stored in a structure of type **TERMINAL**. After **setupterm**( ) returns successfully, the variable **cur_term** (of type **TERMINAL ***) is initialized with all of the information that the *terminfo*(4) boolean, numeric and string variables refer to. The pointer may be saved before calling **setupterm**() again. Further calls to **setupterm**() will allocate new space rather than reuse the space pointed to by **cur_term**.

set_curterm(nterm) *nterm* is of type **TERMINAL** *. **set_curterm**() sets the variable **cur_term** to *nterm*, and makes all of the *terminfo*(4) boolean, numeric and string variables use the values from *nterm*.

del_curterm(oterm) *oterm* is of type **TERMINAL** *. **del_curterm**() frees the space pointed to by *oterm* and makes it available for further use. If *oterm* is the same as **cur_term**, then references to any of the *terminfo*(4) boolean, numeric and string variables thereafter may refer to invalid memory locations until another **setupterm**() has been called.

restartterm(term, fildes, errret)

Similar to **setupterm**(), except that it is called after restoring memory to a previous state; for example, after a call to **scr_restore**(). It assumes that the windows and the input and output options are the same as when memory was saved, but the terminal type and baud rate may be different.

char * tparm(str, p_1, p_2, ..., p_9)

Instantiate the string *str* with parms p_i. A pointer is returned to the result of *str* with the parameters applied.

tputs(str, count, putc)
: Apply padding to the string *str* and output it. *str* must be a *terminfo*(4) string variable or the return value from **tparm**(), **tgetstr**(), **tigetstr**() or **tgoto**(). *count* is the number of lines affected, or **1** if not applicable. *putc* is a *putchar*(3S)-like routine to which the characters are passed, one at a time.

putp(str)
: A routine that calls **tputs** (*str*, **1**, **putchar**).

vidputs(attrs, putc)
: Output a string that puts the terminal in the video attribute mode *attrs*, which is any combination of the attributes listed below. The characters are passed to the *putchar*(3S)-like routine *putc*( ).

vidattr(attrs)
: Similar to **vidputs**(), except that it outputs through *putchar*(3S).

The following routines return the value of the capability corresponding to the character string containing the *terminfo*(4) *capname* passed to them. For example, **rc** = **tigetstr("acsc")** causes the value of **acsc** to be returned in **rc**.

tigetflag(capname)
: The value **−1** is returned if *capname* is not a boolean capability. The value **0** is returned if *capname* is not defined for this terminal.

tigetnum(capname)
: The value **−2** is returned if *capname* is not a numeric capability. The value **−1** is returned if *capname* is not defined for this terminal.

tigetstr(capname)
: The value (char *) **−1** is returned if *capname* is not a string capability. A null value is returned if *capname* is not defined for this terminal.

char * **boolnames**[ ], * **boolcodes**[ ], * **boolfnames**[ ]
char * **numnames**[ ], * **numcodes**[ ], * **numfnames**[ ]
char * **strnames**[ ], * **strcodes**[ ], * **strfnames**[ ]
These null-terminated arrays contain the *capnames*, the *termcap* codes, and the full C names, for each of the *terminfo*(4) variables.

Termcap Emulation

These routines are included as a conversion aid for programs that use the *termcap* library. Their parameters are the same and the routines are emulated using the *terminfo*(4) database.

tgetent(bp, name) Look up *termcap* entry for *name*. The emulation ignores the buffer pointer *bp*.

tgetflag(codename) Get the boolean entry for *codename*.

tgetnum(codename) Get numeric entry for *codename*.

char * **tgetstr**(codename, area)
Return the string entry for *codename*. If *area* is not NULL, then also store it in the buffer pointed to by *area* and advance *area*. **tputs**( ) should be used to output the returned string.

char * **tgoto**(cap, col, row)
Instantiate the parameters into the given capability. The output from this routine is to be passed to **tputs**( ).

tputs(str, affcnt, putc)
See **tputs**( ) above, under "Terminfo-Level Manipulations".

Miscellaneous

traceoff( )
traceon( ) Turn off and on debugging trace output when using the debug version of the *curses* library, */usr/lib/libdcurses.a*. This facility is available only to

customers with a source license.

unctrl(c) This macro expands to a character string which is a printable representation of the character *c*. Control characters are displayed in the ^X notation. Printing characters are displayed as is.

unctrl() is a macro, defined in <**unctrl.h**>, which is automatically included by <**curses.h**>.

char *keyname(c) A character string corresponding to the key *c* is returned.

filter() This routine is one of the few that is to be called before **initscr**() or **newterm**() is called. It arranges things so that *curses* thinks that there is a 1-line screen. *curses* will not use any terminal capabilities that assume that they know what line on the screen the cursor is on.

Use of curscr

The special window **curscr** can be used in only a few routines. If the window argument to **clearok**() is **curscr**, the next call to **wrefresh**() with any window will cause the screen to be cleared and repainted from scratch. If the window argument to **wrefresh**() is **curscr**, the screen is immediately cleared and repainted from scratch. (This is how most programs would implement a "repaint-screen" routine.) The source window argument to **overlay**(), **overwrite**(), and **copywin**() may be **curscr**, in which case the current contents of the virtual terminal screen will be accessed.

Obsolete Calls

Various routines are provided to maintain compatibility in programs written for older versions of the curses library. These routines are all emulated as indicated below.

crmode()	Replaced by **cbreak**().
fixterm()	Replaced by **reset_prog_mode**().
gettmode()	A no-op.
nocrmode()	Replaced by **nocbreak**().
resetterm()	Replaced by **reset_shell_mode**().
saveterm()	Replaced by **def_prog_mode**().
setterm()	Replaced by **setupterm**().

ATTRIBUTES

The following video attributes, defined in <**curses.h**>, can be passed to the routines **wattron**(), **wattroff**(), and **wattrset**(), or OR'ed with the characters passed to **waddch**().

A_STANDOUT	Terminal's best highlighting mode
A_UNDERLINE	Underlining
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_ALTCHARSET	Alternate character set
COLOR_PAIR(n)	Color_pair defined in *n*. (Note that this is a macro).
A_CHARTEXT	Bit-mask to extract character (described under **winch**())
A_ATTRIBUTES	Bit-mask to extract attributes (described under **winch**())
A_NORMAL	Bit-mask to reset all attributes off (for example: **wattrset (win, A_NORMAL)**
A_COLOR	Bit-mask to extract color_pair field informati
PAIR_NUMBER(attrs)	Returns the pair number associated with the COLOR_PAIR(n) attribute (Note that this is a macro).

COLORS

In <**curses.h**> the following macros are defined to have the numeric value shown. These are the default colors. *curses* also assumes that color 0 (zero) is the default background color for all terminals.

COLOR_BLACK	0
COLOR_BLUE	1
COLOR_GREEN	2
COLOR_CYAN	3
COLOR_RED	4
COLOR_MAGENTA	5
COLOR_YELLOW	6
COLOR_WHITE	7

FUNCTION KEYS

The following function keys, defined in <**curses.h**>, might be returned by **wgetch**() if **keypad**() has been enabled. Note that not all of these may be supported on a particular terminal if the terminal does not transmit a unique code when the key is pressed or the definition for the key is not present in the *terminfo*(4) database.

Name	*Value*	*Key name*
KEY_BREAK	0401	break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward + left arrow)
KEY_BACKSPACE	0407	backspace (unreliable)
KEY_F0	0410	Function keys. Space for 64 keys is reserved.
KEY_F(n)	(KEY_F0 + (n))	Formula for f_n.
KEY_DL	0510	Delete line

KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send
KEY_SRESET	0530	soft (partial) reset
KEY_RESET	0531	reset or hard reset
KEY_PRINT	0532	print or copy
KEY_LL	0533	home down or bottom (lower left) keypad is arranged like this: A1 up A3 left B2 right C1 down C3
KEY_A1	0534	Upper left of keypad
KEY_A3	0535	Upper right of keypad
KEY_B2	0536	Center of keypad
KEY_C1	0537	Lower left of keypad
KEY_C3	0540	Lower right of keypad
KEY_BTAB	0541	Back tab key
KEY_BEG	0542	beg(inning) key
KEY_CANCEL	0543	cancel key
KEY_CLOSE	0544	close key
KEY_COMMAND	0545	cmd (command) key
KEY_COPY	0546	copy key
KEY_CREATE	0547	create key
KEY_END	0550	end key

KEY_EXIT	0551	exit key
KEY_FIND	0552	find key
KEY_HELP	0553	help key
KEY_MARK	0554	mark key
KEY_MESSAGE	0555	message key
KEY_MOVE	0556	move key
KEY_NEXT	0557	next object key
KEY_OPEN	0560	open key
KEY_OPTIONS	0561	options key
KEY_PREVIOUS	0562	previous object key
KEY_REDO	0563	redo key
KEY_REFERENCE	0564	ref(erence) key
KEY_REFRESH	0565	refresh key
KEY_REPLACE	0566	replace key
KEY_RESTART	0567	restart key
KEY_RESUME	0570	resume key
KEY_SAVE	0571	save key
KEY_SBEG	0572	shifted beginning key
KEY_SCANCEL	0573	shifted cancel key
KEY_SCOMMAND	0574	shifted command key
KEY_SCOPY	0575	shifted copy key
KEY_SCREATE	0576	shifted create key
KEY_SDC	0577	shifted delete char key
KEY_SDL	0600	shifted delete line key
KEY_SELECT	0601	select key
KEY_SEND	0602	shifted end key
KEY_SEOL	0603	shifted clear line key
KEY_SEXIT	0604	shifted exit key
KEY_SFIND	0605	shifted find key
KEY_SHELP	0606	shifted help key
KEY_SHOME	0607	shifted home key
KEY_SIC	0610	shifted input key
KEY_SLEFT	0611	shifted left arrow key
KEY_SMESSAGE	0612	shifted message key
KEY_SMOVE	0613	shifted move key
KEY_SNEXT	0614	shifted next key

KEY_SOPTIONS	0615	shifted options key
KEY_SPREVIOUS	0616	shifted prev key
KEY_SPRINT	0617	shifted print key
KEY_SREDO	0620	shifted redo key
KEY_SREPLACE	0621	shifted replace key
KEY_SRIGHT	0622	shifted right arrow
KEY_SRSUME	0623	shifted resume key
KEY_SSAVE	0624	shifted save key
KEY_SSUSPEND	0625	shifted suspend key
KEY_SUNDO	0626	shifted undo key
KEY_SUSPEND	0627	suspend key
KEY_UNDO	0630	undo key

LINE GRAPHICS

The following variables may be used to add line-drawing characters to the screen with **waddch()**. When defined for the terminal, the variable will have the **A_ALTCHARSET** bit turned on. Otherwise, the default character listed below will be stored in the variable. The names were chosen to be consistent with the DEC VT100 nomenclature.

Name	*Default*	*Glyph Description*
ACS_ULCORNER	+	upper left corner
ACS_LLCORNER	+	lower left corner
ACS_URCORNER	+	upper right corner
ACS_LRCORNER	+	lower right corner
ACS_RTEE	+	right tee (−|)
ACS_LTEE	+	left tee (|−)
ACS_BTEE	+	bottom tee (⊥)
ACS_TTEE	+	top tee (⊤)
ACS_HLINE	−	horizontal line
ACS_VLINE	|	vertical line
ACS_PLUS	+	plus
ACS_S1	−	scan line 1
ACS_S9	_	scan line 9
ACS_DIAMOND	+	diamond

ACS_CKBOARD	:	checker board (stipple)
ACS_DEGREE	'	degree symbol
ACS_PLMINUS	#	plus/minus
ACS_BULLET	o	bullet
ACS_LARROW	<	arrow pointing left
ACS_RARROW	>	arrow pointing right
ACS_DARROW	v	arrow pointing down
ACS_UARROW	^	arrow pointing up
ACS_BOARD	#	board of squares
ACS_LANTERN	#	lantern symbol
ACS_BLOCK	#	solid square block

DIAGNOSTICS

All routines return the integer **OK** upon successful completion and the integer **ERR** upon failure, unless otherwise noted in the preceding routine descriptions.

All macros return the value of their **w** version, except **getsyx**(), **getyx**(), **getbegyx**(), **getmaxyx**(). For these macros, no useful value is returned.

Routines that return pointers always return **(type *) NULL** on error.

WARNINGS

To use the new f2curses features, use the Release 3.2 version of *curses* on UNIX System V Release 3.1. All programs that ran with Release 2 or Release 3.0 or. Release 3.1 *curses* will also run on UNIX System V Release 3.2. You can link applications with object files based on Release 2 or Release 3.0 or Reelase 3.1 *curses/terminfo* with the Release 3.2 *libcurses.a* library; however, you cannot link applications with object files based on Release 3.2 *curses/terminfo* with the Release 2 or Release 3.0, or Release 3.1 *libcurses.a* library.

Between the time a call to **initscr**() and **endwin**() has been issued, use only the routines in the *curses* library to generate output. Using system calls or the "standard I/O package" (see *stdio*(3S)) for output during that time can cause unpredictable results.

If a pointer passed to a routine as a window argument is null or out of range, the results are undefined (core may be dumped).

SEE ALSO

cc(1), ld(1),

ioctl(2),

putc(3S), scanf(3S), stdio(3S), system(3S), vprintf(3S),

profile(4), term(4), terminfo(4),

varargs(5),

termio(7), tty(7).

curses/terminfo chapter of the *Programmer's Guide*.

This page is intentionally left blank

NAME

cuserid – get character login name of the user

SYNOPSIS

#include <stdio.h>

char * cuserid (s)
char * s;

DESCRIPTION

cuserid generates a character-string representation of the associated with the effective user ID of the process. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least **L_cuserid** characters; the representation is left in this array. The constant **L_cuserid** is defined in the **<stdio.h>** header file.

DIAGNOSTICS

If the login name cannot be found, *cuserid* returns a NULL pointer; if *s* is not a NULL pointer, a null character (**\0**) will be placed at *s* [0]

SEE ALSO

getlogin(3C), getpwent(3C).

This page is intentionally left blank

NAME

dbx – source-level debugger

SYNOPSIS

dbx [**–I** *directory*] [**–c** *file*] [**–** i] [**–** r] [*object*] [*core*]

DESCRIPTION

dbx is a source-level debugger for the Supermax RISC.

The object file used with the debugger is produced by specifying an appropriate option (**–g**) to the compiler. The resulting object file contains symbol table information, including the names of all source files that the compiler translated to create the object file. These source files are accessible from the debugger. If **–g** is not specified, limited debugging is possible.

If a core file exists in the current directory or a coredump is specified, *dbx* can be used to look at the state of the program when it faulted. *dbx* does not support lines greater than 511.

Running dbx

If a *.dbxinit* file resides in the current directory or in the user's home directory, the commands in it are executed when *dbx* is invoked.

When invoked, *dbx* recognizes these command line options:

–I *directory or* **–I***directory*
: Tells *dbx* to look in the specified directory for source files. Multiple directories can be specified by using multiple *–I* options. *dbx* searches for source files in the current directory and in the object file's directory whether or not *–I* is used.

–c *file*
: Selects a command file other than *.dbxinit*.

–i
: Uses interactive mode. This option does not treat #s as comments in a file. It prompts for source even when it reads from a file. With this option, *dbx* also has extra formatting as if for a terminal.

−r Runs the object file immediately.

The *dbx* monitor offers powerful command line editing. For a full description of these editing features, see *csh*(1).

Multiple commands can be specified on the same command line by separating them with a semicolon (;). If the user types a string and presses the stop character usually (^z; see *stty*(1), *dbx* tries to complete a symbol name from the program that matches the string.

The Monitor

These commands control the *dbx* monitor:

![*string*] [*integer*] [−*integer*]
Specifies a command from the history list.

help Prints a list of *dbx* commands, using the UNIX system 'more' command to display the list.

history Prints the items from the history list. The default is 20.

quit[!] Exit dbx after verification. If '!' is specified, verification is not required.

Controlling dbx

alias [*name(arg1,...argN)"string"*]
Lists all existing aliases, or, if an argument is specified, defines a new alias.

unalias *alias command name*
Removes the specified alias.

delete *expression1,...expressionN*

delete *all* Deletes the specified item from the status list. The argument *all* deletes all items from the status list.

playback input [*file*]
Replays commands that were saved with the record input commands in a text file.

playback output [*file*]
Replays debugger output that was saved with the record output command.

record input [*file*]
Records all commands typed to dbx.

record output [*file*]
Records all dbx output.

sh [*shell command*]
Calls a shell from dbx or executes a shell command.

status
Lists currently set stop, record, and trace commands.

tagvalue (*tagname*)
Returns the value of *tagname*. If the tags extends to more than one line, or if it contains arguments, an error occurs. *tagvalue* can be used in any expression.

set [*variable* = *expression*]
Lists existing debugger variables and their values. This command can also be used to assign a new value to an existing variable or to define a new variable.

unset *variable*
Removes the setting of a specified debugger variable.

Examining Source

/regular expression
Searches ahead in the source code for the regular expression.

?regular expression
Searches back in the source code for the regular expression.

edit [*file*] Calls an editor from dbx.

file [*file*] Prints the current file name, or, if a file name is specified, this command changes the current file to the specified file.

func [*expression*] [*procedure*]
Moves to the specified procedure (activation level), or, if an expression or procedure is not specified, prints the current activation level.

list [*expression:integer*]

list [*expression*]
Lists the specified lines. The default is 10 lines.

tag *tagname* Sets the current file/line to the location specified by *tagname*. Operations are similar to tge tag operations in *vi*(1).

use [*directory1 . . . directoryN*]
Lists source directories, or, if a directory name is specified, this command substitutes the new directories for the previous list.

whatis *variable*
Prints the type declaration for the specified name.

which *variable*
Finds the variable name currently being used.

whereis *variable*
Prints all qualifications (the scopes) of the specified variable name.

Controlling Programs

assign *expression1 = expression2*
Assigns the specified expression to a specified program variable.

[*n*] **cont** [*signal*]

cont [*signal*] **to** *line*

cont [*signal*] **in** *procedure*

Continues executing a program after a breakpoint. *n* breakpoints are ignored if *n* is specified before stepping. IOIf specified, *signal* is delivered to the processing being debugged.

goto *line* — Goes to the specified line in the source.

next [*integer*] — Steps over the specified number of lines. The default is one. This command does not step into procedures.

rerun [*arg1 ... argN*] [<*file1*] [>*file2*]

rerun [*arg1 ... argN*] [<*file1*] [>&*file2*]

Reruns the program, using the same arguments that were specified to the run command. If new arguments are specified, rerun uses those arguments.

run [*arg1 ... argN*] [<*file1*] [>*file2*]

run [*arg1 ... argN*] [<*file1*] [>&*file2*]

Runs the program with the specified arguments.

return [*procedure*]

Continues executing until the procedire returns. If a procedure is not specified, dbx assumes the next procedure.

step [*integer*] — Steps the specified number of lines. This command steps into procedures. The default is one line.

Setting Breakpoints

catch [*signal*]

Lists al signals that dbx catches, or, if an argument is specified, adds a new signal to the catch list.

ignore [*signal*]
Lists all signals that dbx does not catch. If a signal is specified, this command adds the signal to the ignore list.

stop [*variable*]

stop [*variable*] **at** *line* [**if** *expression*]

stop [*variable*] **in** *procedure* [**if** *expression*]

stop [*variable*] **if** *expression*
Sets the breakpoint at the specified point.

trace *variable* [**at** *line*] [*if expression*]

trace *variable* [**in** *procedure*] [*if expression*]
Traces the specified variable.

when [*variable*] [**at** *line*] {*command list*}

when [*variable*] [**in** *procedure*] {*command list*}
Executes the specified dbx comma separated command list.

Examining Program State

dump [*procedure*] [.]
Prints variable information about procedure. If a dot (.) is specified, this command prints global variable information on all procedures in the stack and the variables of those procedures.

down [*expression*]
Moves down the specified number of activation levels in the stack. The default is one level.

up [*expression*]
Moves up the specified number of activation levels in the stack. The default is one level.

print *expression1, ... expressionN*
Prints the value of the specified expression. If *expression* is a dbx keyword, it must be enclosed in parantheses. For example, to print out a variable called 'output' (which is also a

variable in the playback and record commands) you must type:

print (output)

printf "*string*", *expression1, ... expressionN*
: Prints the value of the specified expression, using C language string formatting. As in the print command, if *expression* is a dbx keyword, you must enclose it within parantheses.

printregs
: Prints all register values.

where
: Does a stack trace, which shows the current activation levels.

where *n*
: Prints out only the top *n* levels of the stack.

Debugging at the Machine Level

[*n*] **conti** [*signal*]

conti [*signal*] **to** *address*

conti [*signal*] **in** *procedure*
: Continues executing assembly code after a breakpoint. *n* breakpoints are ignored if *n* is specified before stepping. If specified, *signal* is delivered to the processing being debugged.

nexti [*integer*]
: Steps over the specified number of machine instructions. The default is one. This command does not step into procedures.

stepi [*integer*]
: Steps the specified number of machine instructions. This command steps into procedures. The default is one instruction.

stopi [*variable*] **at** [*address*] [**at** *address* [**if** *expression*]

stopi [*variable*] **in** *procedure* [**if** *expression*]

stopi [*variable*] **if** *expression*
: Sets the breakpoint in the machine code at the specified point.

tracei *variable* **at** *address* [**at** *address* **if** *expression*]

tracei *variable* **in** *procedure* [**at** *address* **if** *expression*]
: Traces the specified variable in machcine instructions.

wheni [*variable*] [**at** *address*] {*command*}

wheni [*variable*] [**in** *procedure*] {*command*}
: Executes the specified dbx comma separated command list.

address[?]/**<count> <mode>**
: Searching forward (or backward, if ? is specified), prints the contents *address*, or disassembles the code for the instruction *address*; *count* is the number of items to be printed at the specified address. *mode* is one of the characters in the following table producing the indicated result:

d	Print a short word in decimal.
D	Print a long word in decimal.
o	Print a short word in octal.
O	Print a long word in octal.
x	Print a short word in hexadecimal.
X	Print a long word in hexadecimal.
b	Print a byte in octal.
c	Print a bite as a character.
s	Print a string of characters that ends in a null.
f	Print a single precision real number.
g	Print a double precision real number.
i	Print machine instructions.
n	Prints data in typed format.

address / < *countL* > < *value* > < *mask* >
: Searches for a 32-bit word starting at the specified *address*; *count* specifies the number of word to process in the search; an address is printed when the word *at address*, after an AND operation with *mask*, is equal to *value*.

Predefined dbx Variables:

The debugger has these predefined variables:

$addfmt Specifies the format for addresses. This can be set to any specification that a C 'printf' statement can format. The default is zero.

$byteaccess Same as **$addrfmt**.

$casesence When set to a nonzero value, specifies that uppercase and lowercase letters be taken into consideration during a search. When set to 0, the case is ignored. The default i 0.

$curevent Shows the last even number as seen in the status feature. Set only by dbx.

$curline Specifies the current line. Set only by dbx.

$curscrline Shows the last line listed plus 1. Set only by dbx.

$curpc Specifies the current address. Used with the *wi* and *li* aliases.

$datacache Caches information from the data space so that dbx must access data space only once. To debug the operating system, set this variable to 0; otherwise set it to a nonzero value. The default is 1.

$debugflag For internal use by dbx.

$defin For internal use by dbx.

$defout For internal use by dbx.

$dispix For use when debugging pixie code. When set to 0, machine code is showed while debugging. When set to 1, pixie code is shown. The default is 0.

$hexchars Output characters are printed in hexadecimal format (set, unset).

$hexin Specifies that inout constants are hexadecimal.

$hexints When set to a nonzero value, changes the default output constants toi hexadecimal. Overrides *$octints*.

$hexstrings When set to 1, specifies that all strings are printed in hexadecimal; when set to 0, strings are printed in character format.

$historyevent
Shows the current history line.

$lines Number of lines for history. The default is 20.

$listwindow Specifies how many lines the *list* command prints.

$main Specifies the name of the procedure that dbx will start with. This can be set to any procedure. The default is 'main'.

$maxstrlen Specifies how many characters of a string that dbx prints for pointers to strings. The default is 128.

$octin When set to nonzero, changes the default input constants to octal. When set, *$hexint* overrides this setting.

$octints Output integers are printed octal format (set, unset).

$page Specifies whether to page long information. A nonzero value turns on paging; a 0 turns it off. The default is 1.

$pagewindow
Specifies how many lines print when information runs longer than one screen. This can be changed to match the number of lines on any terminal. If set to 0, this variable assumes one line. The default is 22, leaving space for continuation query.

$pdbxport Port name from */etc/remote*[*pdbx*] used to connect to target machine for pdbx.

$printwhilestep
For use with the step[*n*] and stepi[*n*] instructions. A nonzero integer specifies that all *n* lines and/or instructions should be printed out. A zero specifies that only the last line and/or instruction should be printed out. The default is zero.

$pimode Prints input when used with the *playback input* command. The default is 0.

$printdata When set to a nonzero value, the contents of registers used are printed next to each instruction displayed. The default is 0.

$printwide When set to a nonzero value, the contents of variables are printed in a horizontal format. The default is 0.

$prompt Sets the prompt for dbx.

$readtextfile When set to 1, dbx tries to read instructions from the object file rather than the process. dbx executes faster when debugging remotely using the System Programmer's Package. This variable should always be set to 0 when the process being debugged copies in code during the debugging process. The default is 1.

$regstyle A zero value causes registers to be printed out in their normal *r* format (*r0,r1, ... r31*). A nonzero value causes the registers to be printed out in a special format (*zero, at, v0, v1, ...*) commonly used in debugging programs written in assembly language.

$repeatmode
When set to a nonzero value, after pressing the RETURN key (for an empty line), the last command is repeated. The default is 1.

$rimode When set to a nonzero value, input is recorded while recording output. The default is 0.

$sigtramp Tells dbx the name of the code called by the system to invoke user signal handlers. This variable is set to *sigtramp* system running under RISC/os.

$tagfile Contains a filename, indicating the file in which the tag command and the tabvalue macro are to search for tags.

Predefined dbx Aliases

The debugger has these predefined aliases:

?	Prints a list of all dbx commands.
a	Assigns a value to a program variable.
b	Sets a breakpoint at a specified line.
bp	Stops in a specified procedure.
c	Continues program execution after a breakpoint.
d	Deletes the specified item from the status list.
e	Looks at the specified line.
f	Moves to the specified activation level on the stack.

g Goes to the specified line and begins executing the program there.

h Lists all items currently on the history list.

j Shows what items are on the status list.

l Lists the next 10 lines of source code.

li Lists the next 10 machine instructions.

n or S Step over the specified number of lines without stepping into procedure calls.

ni or Si
Step over the specified number of assembly code instructions without stepping into procedure calls.

p Prints the value of the specified expression or variable.

pd Prints the value of the specified expression or variable in decimal.

pi Replays dbx commands that were saved with the record input format.

po Prints the value of the specified expression or variable in octal.

pr Prints values for all registers.

px Prints the value for the specified variable or expression in hexadecimal.

q Ends the debugging session.

r Runs the program again with the same arguments that were specified with the 'run' command.

ri Records in a file every command typed.

ro Records all debugger output in the specified file.

s Steps the next number of specified lines.

si Steps the next number of specified lines of assembly code instructions.

t Does a stack trace.

u Lists the previous 10 lines.

w Lists the 5 lines preceding and following the current line.

W Lists the 10 lines preceding and following the current line.

wi Lists the 5 machine instructions preceding and following the machine instruction.

NOTE:

In order to use all facilities in *dbx* it is important that the word **LINEEDIT=** is placed in the environment:

```
LINEEDIT=
export LINEEDIT
```

SEE ALSO

dbx in the *Programmers Guide.*

NAME

dial – establish an out-going terminal line connection

SYNOPSIS

#include <dial.h>

int dial (call)
CALL call;

void undial (fd)
int fd;

DESCRIPTION

dial returns a file-descriptor for a terminal line open for read/write. The argument to *dial* is a CALL structure (defined in the <*dial.h*> header file).

When finished with the terminal line, the calling program must invoke *undial* to release the semaphore that has been set during the allocation of the terminal device.

The definition of CALL in the <*dial.h*> header file is:

```
typedef struct {
   struct termio
   *attr;/* pointer to termio attribute struct */
   int   baud;    /* transmission data rate */
   int   speed;   /* 212A modem: low=300, high=1200 */
   char  *line;   /* device name for out-going line */
   char  *telno;  /* pointer to tel-no digits string */
   int   modem;   /* specify modem control for direct lines */
   char  *device; /* unused */
   int   dev_len; /* unused */
} CALL;
```

The CALL element *speed* is intended only for use with an outgoing dialed call, in which case its value should be either 300 or 1200 to identify the 113A modem, or the high- or low-speed setting on the 212A modem. Note that the 113A modem or the low-speed setting of the 212A modem will transmit at any rate between 0 and 300 bits per second. However, the high-speed setting of the 212A modem transmits and receives at 1200 bits per second only. The CALL element *baud* is for the desired transmission baud rate. For example, one might set *baud* to 110 and *speed* to 300 (or 1200). However, if *speed* is set to 1200, *baud* must be set to high (1200).

If the desired terminal line is a direct line, a string pointer to its device-name should be placed in the *line* element in the CALL structure. Legal values for such terminal device names are kept in the *Devices* file. In this case, the value of the *baud* element should be set to -1. This will cause **dial** to determine the correct value from the *Devices* file.

The *telno* element is for a pointer to a character string representing the telephone number to be dialed. Such numbers may consist only of these characters:

0-9	dial 0-9
*	dial *
#	dial #
=	wait for secondary dail tone
–	delay for approximately 4 seconds

The CALL element *modem* is used to specify modem control for direct lines. This element should be non-zero if modem control is required. The CALL element *attr* is a pointer to a *termio* structure, as defined in the *termio.h* header file. A NULL value for this pointer element may be passed to the *dial* function, but if such a structure is included, the elements specified in it will be set for the outgoing terminal line before the connection is established. This is often important for certain attributes such as parity and baud-rate.

The CALL elements device and *dev_len* are no longer used. They are retained in the CALL structure for compatibility reasons.

FILES

/usr/lib/uucp/Devices
/usr/lib/uucp/Systems
/usr/lib/uucp/Sysfiles
/usr/lib/uucp/Dialers
/usr/lib/uucp/Devconfig
/usr/spool/uucp/LCK..*tty-device*

SEE ALSO

uucp(1C), alarm(2), read(2), write(2) and termio(7).

DIAGNOSTICS

On failure, a negative value indicating the reason for the failure will be returned. Mnemonics for these negative indices as listed here are defined in the <*dial.h*> header file.

INTRPT	−1	/* interrupt occurred */
D_HUNG	−2	/* dialer hung (no return from write) */
NO_ANS	−3	/* no answer within 10 seconds */
ILL_BD	−4	/* illegal baud-rate */
A_PROB	−5	/* acu problem (open() failure) */
L_PROB	−6	/* line problem (open() failure) */
NO_Ldv	−7	/* can't open *Devices* file */
DV_NT_A	−8	/* requested device not available */
DV_NT_K	−9	/* requested device not known */
NO_BD_A	−10	/* no device available at requested baud */
NO_BD_K	−11	/* no device known at requested baud */
DV_NT_E	−12	/* requested speed does not match */
BAD_SYS	−13	/* system not in *Systems* file */

NOTE

The program must be linked with the *libdial.a* archive; *cc* must be called with the **−ldial** option.

WARNINGS

The R3000 version of the *dial* library function is not compatible with Basic Networking Utilities on UNIX System V Release 2.0.

Including the <*dial.h*> header file automatically includes the <*termio.h*> header file.

The above routine uses <*stdio.h*>, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

An *alarm*(2) system call for 3600 seconds is made (and caught) within the *dial* module for the purpose of "touching" the *LCK..* file and constitutes the device allocation semaphore for the terminal device. Otherwise, *uucp*(1C) may simply delete the *LCK..* entry on its 90-minute clean-up rounds. The alarm may go off while the user program is in a *read*(2) or *write*(2) system call, causing an apparent error return. If the user program expects to be around for an hour or more, error returns from *read*s should be checked for **(errno= =EINTR)**, and the *read* possibly reissued.

NAME

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

SYNOPSIS

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

DIR * opendir (filename)
char * filename;

struct dirent * readdir (dirp)
DIR * dirp;

long telldir (dirp)
DIR * dirp;

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

void rewinddir (dirp)
DIR *dirp;

int 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*(3C) enough memory to hold a DIR structure or a buffer for the directory entries.

readdir returns a pointer to the next active directory entry, and positions the directory stream at the next entry. No inactive entries are returned. It returns NULL upon reaching the end of the directory or upon detecting an invalid location in the directory.

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 from which *loc* was obtained was performed.

rewinddir resets the position of the named *directory stream* to the beginning of the directory.

closedir closes the named *directory stream* and frees the DIR structure.

The following errors can occur as a result of these operations.

opendir:

[EACCES]	A component of *filename* denies search permission, or read permission is denied for dirname.
[EFAULT]	*filename* points outside the allocated address space.
[EMFILE]	The maximum number of file descriptors are currently open.
[ENAMETOOLONG]	The length of the *filename* argument exceeds {PATH_MAX}, or the length of a *filename* component exceeds {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOENT]	The dirname argument points to the name of a file which does not exist, or to an empty string.
[ENOTDIR]	A component of *filename* is not a directory.

readdir:

[EBADF] The file descriptor determined by the DIR stream is no longer valid. This results if the DIR stream has been closed.

[ENOENT] The current file pointer for the directory is not located at a valid entry.

telldir, *seekdir*, and *closedir*:

[EBADF] The file descriptor determined by the DIR stream is no longer valid. This results if the DIR stream has been closed.

EXAMPLE

Sample code which searches a directory for entry *name*:

```
dirp = opendir( "." );
while ( (dp = readdir( dirp )) != NULL )
        if ( strcmp( dp->d_name, name ) == 0 )
                {
                closedir( dirp );
                return FOUND;
                }
closedir( dirp );
return NOT_FOUND;
```

SEE ALSO

getdents(2), dirent(4).

WARNINGS

rewinddir is implemented as a macro, so its function address cannot be taken.

This page is intentionally left blank

NAME

drand48, erand48, jrand48, lrand48, nrand48, mrand48, srand48, seed48, lcong48 – generate uniformly distributed pseudo-random numbers

SYNOPSIS

double drand48 ( )

double erand48 (xsubi)
unsigned short xsubi[3];

long lrand48 ( )

long nrand48 (xsubi)
unsigned short xsubi[3];

long mrand48 ( )

long jrand48 (xsubi)
unsigned short xsubi[3];

void srand48 (seedval)
long seedval;

unsigned short * seed48 (seed16v)
unsigned short seed16v[3];

void lcong48 (param)
unsigned short param[7];

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions *drand48* and *erand48* return non-negative double-precision floating-point values uniformly distributed over the interval $[0.0, 1.0)$.

Functions *lrand48* and *nrand48* return non-negative long integers uniformly distributed over the interval $[0, 2^{31})$.

Functions *mrand48* and *jrand48* return signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$.

Functions *srand48, seed48* and *lcong48* are initialization entry points, one of which should be invoked before either *drand48, lrand48* or *mrand48* is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if *drand48, lrand48* or *mrand48* is called without a prior call to an initialization entry point.)

Functions *erand48, nrand48* and *jrand48* do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula

$$X_{n+1} = (aX_n + c)_{\mathrm{mod}\ m} \qquad n \geq 0.$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless *lcong48* has been invoked, the multiplier value a and the addend value c are given by

$$a = \mathrm{5DEECE66D}_{16} = 273673163155_8$$
$$c = \mathrm{B}_{16} = 13_8.$$

The value returned by any of the functions *drand48, erand48, lrand48, nrand48, mrand48* or *jrand48* is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of X_i and transformed into the returned value.

The functions *drand48, lrand48* and *mrand48* store the last 48-bit X_i generated in an internal buffer, and must be initialized prior to being invoked. The functions *erand48, nrand48* and *jrand48* require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked.

These routines do not have to be initialized; the calling program must place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, functions *erand48, nrand48* and *jrand48* allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers, i.e., the sequence of numbers in each stream will *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function *srand48* sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer function *seed48* sets the value of X_i to the 48-bit value specified in the argument array. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by *seed48,* and a pointer to this buffer is the value returned by *seed48.* This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time – use the pointer to get at and store the last X_i value, and then use this value to reinitialize via *seed48* when the program is restarted.

The initialization function *lcong48* allows the user to specify the initial X_i, the multiplier value a, and the addend value c. Argument array elements *param[0-2]* specify X_i, *param[3-5]* specify the multiplier a, and *param[6]* specifies the 16-bit addend c. After *lcong48* has been called, a subsequent call to either *srand48* or *seed48* will restore the "standard" multiplier and addend values, a and c, specified on the previous page.

SEE ALSO

rand(3C).

This page is intentionally left blank

NAME

dup – duplicate an open file descriptor

SYNOPSIS

int dup(fildes)
  int fildes;

DESCRIPTION

fildes is a file-descriptor obtained from a *creat*, *dup*, *fcntl*, *open* or *pipe* system call. *dup* returns a new file-descriptor having the following in common with the original:

- Same open file (or pipe).
- Same file-pointer (that is, both file-descriptors share one file-pointer).
- Same access mode (read, write, read/write etc.).

The new file-descriptor is set to remain open across calls to the *exec*(2) routines [see *fcntl*(2)].

The file-descriptor returned is the lowest one available.

RETURN VALUE

If successful, the function *dup* will return a non-negative integer, namely the file-descriptor; otherwise, it will return −1 and **errno** will indicate the error.

ERRORS

Under the following conditions *dup* will fail and will set **errno** to:

[EBADF]	If fildes is not a valid open file-descriptor.
[EMFILE]	If OPEN_MAX file-descriptors are currently open in the calling process.

SEE ALSO

close(2), creat(2), exec(2), fcntl(2), open(2), pipe(2), dup2(3C), lockf(3C).

This page is intentionally left blank

NAME

dup2 – duplicate an open file descriptor

SYNOPSIS

int dup2 (fildes, fildes2)
int fildes, fildes2;

DESCRIPTION

fildes is a file descriptor referring to an open file, and *fildes2* is a non-negative integer less than OPEN_MAX. *dup2* causes *fildes2* to refer to the same file as *fildes*. If *fildes2* already referred to an open file, it is closed first.

dup2 will fail if one or more of the following are true:

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

[EMFILE] OPEN_MAX file descriptors are currently open.

SEE ALSO

close(2), creat(2), dup(2), exec(2), fcntl(2), open(2), pipe(2), lockf(3C).

DIAGNOSTICS

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

This page is intentionally left blank

NAME

ecvt, fcvt, gcvt − convert floating-point number to string

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;
int ndigit;
char ∗ buf;

DESCRIPTION

ecvt converts *value* to a null-terminated string of *ndigit* digits and returns a pointer thereto. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. 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). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero.

fcvt is identical to *ecvt*, except that the correct digit has been rounded for printf "%f" (FORTRAN F-format) output of the number of digits specified by *ndigit*.

gcvt converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F-format if possible, otherwise E-format, ready for printing. A minus sign, if there is one, or a decimal point will be included as part of the returned string. Trailing zeros are suppressed.

SEE ALSO

printf(3S).

BUGS

The values returned by *ecvt* and *fcvt* point to a single static data array whose content is overwritten by each call.

NAME

edit – update a line of text from a terminal

SYNOPSIS

int edit (fildes, buf, nbyte, curoff)
int fildes;
char * buf;
unsigned nbyte, curoff;

DESCRIPTION

edit reads a line of text at most *nbyte* long (excluding the final new-line character) from the file associated with *fildes* into the buffer pointed to by *buf*.

The argument *fildes* is an open file-descriptor [see *file-descriptor* in the introduction, *intro*(2&3)].

The function is primarily intended to be used on terminals. When operating on a terminal, the contents of the character buffer pointed to by *buf* will be output to the terminal, and the cursor will be left at offset *curoff* from the first character in the buffer. After this, the operator, using the normal line editing commands, may change the contents of the buffer and terminate input as with *read*(2). The operator is unable to move the cursor beyond the end of the buffer, the size of which is *nbyte* bytes.

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

[EBADF]	*fildes* is not a valid file-descriptor.
[EINTR]	A signal was caught during the operation.
[EIO]	A physical I/O error has occurred.
[ENXIO]	The device associated with the file-descriptor is a special file and the value of the file-pointer is out of range.
[EAGAIN]	The file is an ordinary file, enforcement-mode file and record locking was set, O_NDELAY was set, and there was a blocking write-lock.

[EDEADLK]	The read was going to sleep and cause a deadlock situation to occur.
[ENOLCK]	The system record-lock table was full, so that the read could not go to sleep.

SEE ALSO

read(2).

DIAGNOSTICS

If successful, the functions *edit* will return a non-negative integer indicating the number of bytes actually read (excluding the final new-line character); if an end-of-file condition is met, the functions will return −2; otherwise, they will return −1 and *errno* is set to indicate the error.

NOTE

The program must be loaded with the library **libdde.a**.

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 (the first location beyond the data) coincides with *end*, but the program break may be reset by the routines of *brk*(2), *malloc*(3C), standard input/output [*stdio*(3S)], the profile (**–p**) option of *cc*(1), and so on. Thus, the current value of the program break should be determined by **sbrk (char ∗)(0)** [see *brk*(2)].

SEE ALSO

cc(1), brk(2), malloc(3C), stdio(3S).

This page is intentionally left blank

NAME

erf, erfc – error function and complementary error function

SYNOPSIS

```
#include <math.h>

double erf (x)
  double x;

double erfc (x)
  double x;
```

DESCRIPTION

erf returns the error function of x, defined as $\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$.

erfc, which returns 1.0 − *erf(x)*, is provided because of the extreme loss of relative accuracy if *erf(x)* is called for large x and the result subtracted from 1.0 (e.g., for $x = 5$, 12 places are lost).

If x is NaN, NaN is returned and *errno* is set to EDOM.

SEE ALSO

exp(3M).

This page is intentionally left blank

NAME

exec: execl, execv, execle, execve, execlp, execvp — execute a file

SYNOPSIS

extern char * * environ

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

int execv (path, argv)
char * path, * argv[ ];

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

int execve (path, argv, envp)
char * path, * argv[ ], * envp[ ];

int execlp (file, arg0, arg1, ..., argn, 0)
char * file, * arg0, * arg1, ..., * argn;

int execvp (file, argv)
char * file, * argv[ ];

DESCRIPTION

exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the *new process image file*. There can be no return from a successful *exec* because the calling process is overlaid by the new process.

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, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. 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.

path points to a path name that identifies the new process image file.

file points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH =" [see *environ*(5)]. The environment is supplied by the shell [see *sh*(1)].

If the process image file is not a valid executable object, the *execlp*( ) and *execvp*( ) functions use the contents of that file as standard input to a command interpreter conforming to *system*( ). In this case the command interpreter becomes the new process image.

arg0, *argl*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process image. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component).

argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process image. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). *argv* is terminated by a null pointer.

envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. *envp* is terminated by a null pointer. For *execl* and *execv*, the C run-time start-off routine places a pointer to the environment of the calling process in the global cell:

extern char * * environ;

and it is used to pass the environment of the calling process to the new process image.

The operating system will choose an MCU on which the new process will bee executed. The MCU chosen is the one that numerically follows the MCU chosen by the last *exec* call, with the following modifications: Only MCUs that are physically present on the computer are chosen. Only the MCUs specified in the calling process' *MCU-mask* [see *mcumask*(2)] will be chosen. The selection may be further restricted by preceding the *exec* call by

set_parm(mask, −1, −1);

If *mask* is 0, the new process will run on the same MCU as the calling process. Otherwise, *mask* is bitwise AND'ed with the calling process' MCU-mask to create a mask specifying which MCUs the operating system may choose between.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl*(2). For those file descriptors that remain open, the file pointer is unchanged.

Signals set to terminate the calling process will be set to terminate the new process. Signals set to be ignored by the calling process will be set to be ignored by the new process. Signals set to be caught by the calling process will be set to terminate new process; see *signal*(2).

For signals set by *sigset*(2), *exec* will ensure that the new process has the same system signal action for each signal type whose action is SIG_DFL, SIG_IGN, or SIG_HOLD as the calling process. However, if the action is to catch the signal, then the action will be reset to SIG_DFL, and any pending signal for this type will be held.

If the set-user-ID mode bit of the new process file is set [see *chmod*(2)], *exec* sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process.

The shared memory segments attached to the calling process will not be attached to the new process [see *shmop*(2) and *par_att*(2)].

Profiling is disabled for the new process; see *profil*(2).

All active asynchronous I/O operations started by the calling process are aborted.

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

current working directory
file mode creation mask [see *umask*(2)]
file size limit [see *ulimit*(2)]
file-locks [see *fcntl*(2) and *lockf*(3C)]
nice value [see *nice*(2)]
parent process ID
pending signal [see *sigpending*(2)]
process signal mask [see *sigprogmask*(2)]
process ID
process group ID
real group ID
real user ID
root directory
semadj values [see *semop*(2)]
time left until an alarm clock signal [see *alarm*(2)]
trace flag [see *ptrace*(2) request 0]
tty group ID [see *exit*(2) and *signal*(2)]
tms_utime, *tms_stime*, *tms_cutime*, and *tms_cstime* [see *times*(2)]
MCU mask [see *mcumask*(2)]

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

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

[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 is not an ordinary file.
[EAGAIN]	All local process control blocks on the destination MCU are in use.
[EFAULT]	*path*, *argv*, or *envp* point to an illegal address.
[EINTR]	A signal was caught during the *exec* system call.
[EINVAL]	The *exec* is preceded by a call to *set_parm* with a *mask* that specifies no legal MCU.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[ENAMETOOLONG]	The length of the *path* or *file* arguments, or an element of the environment variable PATH prefixed to a file, exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} and {_POSIX_NO_TRUNC} is in effect for that file.
[ENOENT]	One or more components of the new process path name of the file do not exist.
[ENOEXEC]	The exec is not an *execlp* or *execvp*, and the new process file has the appropriate access permission but an invalid magic number in its header.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[ENOMEM]	The new process requires more memory than is allowed by the system-imposed maximum MAXMEM.

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

[ETXTBSY] The new process file is a pure procedure (shared text) file that is currently open for writing by some process.

If an error is detected after the calling process has disappeared, the new process is terminated with a SIGKILL signal. A typical error of this kind would be the inability to load the program from the new process file.

SEE ALSO

sh(1), alarm(2), exit(2), fcntl(2), fork(2), nice(2), ptrace(2), semop(2), signal(2), sigset(2), times(2), ulimit(2), umask(2), lockf(3C), a.out(4), environ(5).

DIAGNOSTICS

If *exec* returns to the calling process an error has occurred; the return value will be −1 and *errno* will be set to indicate the error.

NAME

exit, _exit – terminate process

SYNOPSIS

#include <stdlib.h>

void exit (status)
int status;

void _exit (status)
int status;

DESCRIPTION

exit terminates the calling process with the following consequences:

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

All the active asynchronous I/O operations started by the calling process are aborted.

If the parent process of the calling process is executing a *wait*, it is notified of the calling process's termination and the low order eight bits (i.e., bits 0xff) of *status* are made available to it [see *wait*(2)].

If the parent process of the calling process is not executing a *wait*, the calling process is transformed into a zombie process. A *zombie process* is an incative process that has no space allocated to it, and it will be deleted at some later time when its parent executes a *wait*(2) routine or dies.

The parent process ID of all of the calling processes' existing child processes and zombie processes is set to 1. This means the initialization process [see *intro*(2)] inherits each of these processes.

Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1.

For each semaphore for which the calling process has set a *semadj* value [see *semop*(2)], that semadj value is added to the semval of the specified semaphore.

If the process has a process, text, or data lock, an *unlock* is performed [see *plock* (2)].

An accounting record is written on the accounting file if the system's accounting routine is enabled [see *acct* (2)].

If the process ID, tty group ID, and process group ID of the calling process are equal, the **SIGHUP** signal is sent to each process that has a process group ID equal to that of the calling process.

A death of child signal is sent to the parent.

The C function *exit* may cause cleanup actions before the process exits. The function *_exit* circumvents all cleanup.

The C function *exit*(3C) calls any functions registered through the *atexit* function in the reverse order of their registration. The function *_exit* circumvents all such functions and cleanup.

The symbols EXIT_SUCCESS and EXIT_FAILURE are defined in **stdlib.h** and may be used as the value of *status* to indicate successful or unsuccessful termination, respectively.

SEE ALSO

acct(2), intro(2), plock(2), semop(2), signal(2), sigset(2), wait(2).

DIAGNOSTICS

None. There can be no return from an *exit* system call.

NAME

exp, log, log10, pow, sqrt – exponential, logarithm, power, square root functions

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 e^x.

log returns the natural logarithm of x. The value of x must be positive.

log10 returns the logarithm base ten of x. The value of x must be positive.

pow returns x^y. If x is zero, y must be positive. If x is negative, y must be an integer.

sqrt returns the non-negative square root of x. The value of x may not be negative.

SEE ALSO

hypot(3M), sinh(3M).

DIAGNOSTICS

exp returns **HUGE_VAL** when the correct value would overflow, or 0 when the correct value would underflow, and sets *errno* to **ERANGE**.

For all functions, if x is NaN, NaN is returned and *errno* is set to EDOM.

log and *log10* return **−HUGE_VAL** and set *errno* to **EDOM** when x is non-positive.

pow returns 1.0 if x and y is zero. *pow* returns 0 and sets *errno* to **EDOM** when x is 0 and y is non-positive, or when x is negative and y is not an integer. When the correct value for *pow* would overflow or underflow, *pow* returns ±**HUGE_VAL** or 0 respectively, and sets *errno* to **ERANGE**.

sqrt returns 0 and sets *errno* to **EDOM** when x is negative.

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

```
#include <stdio.h>

int fclose (stream)
 FILE *stream;

int fflush (stream)
 FILE *stream;
```

DESCRIPTION

fclose causes any buffered data for the named *stream* to be written out, and the *stream* to be closed. It marks for update the *st_ctime* and *st_mtime* fields of the underlaying file, if the stream was writable, and if buffered data had not been written to the file yet.

fclose is performed automatically for all open files upon calling *exit*(2).

fflush causes any buffered data for the named *stream* to be written to that file. The *stream* remains open. The *st_ctime* and *st_mtime* fields are marked for update.

SEE ALSO

close(2), exit(2), fopen(3S), setbuf(3S), stdio(3S).

DIAGNOSTICS

These functions return 0 for success, and **EOF** if any error is detected and *errno* is set to:

[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor underlaying *stream* andhe process would be delayed in the write operation.
[EBADF]	The file descriptor underlaying *stream* is not valid.
[EFBIG]	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size. (See *ulimit*(2)).

[EINTR] The *fflush*(2) function was interrupted by a signal.

[EIO] The implementation supports job control, the process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU and the process group of the process is orphaned. This error may also be returned under implementation defined conditions.

[ENOSPC] There was no free space remaining on the device containing the file.

[EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal will also be sent to the process.

NAME

fcntl – file control

SYNOPSIS

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

int fcntl (fildes, cmd, arg)
 int fildes, cmd, arg;
```

DESCRIPTION

fcntl provides for control over open files. *Fildes* is an open file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

The commands available are:

F_DUPFD Return a new file descriptor as follows:

Lowest numbered available file descriptor greater than or equal to *arg*.

Same open file (or pipe) as the original file.

Same file pointer as the original file (i.e., both file descriptors share one file pointer).

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 *exec*(2) system calls.

F_GETFD Get the close-on-exec flag FD_CLOEXEC associated with the file descriptor *fildes*. 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 FD_CLOEXEC flag associated with *fildes* to the low-order bit of *arg* (0 or 1 as above).

F_GETFL — Get *file* status flags.

F_SETFL — Set *file* status flags to *arg*. Only certain flags can be set [see *fcntl*(5)].

The following commands are used for file locking and record locking (byte locking). Locks may be placed on an entire file or segments of a file. If enforcement-mode file and record locking is in effect [see *chmod*(2)] a lock will prevent read and write operations that are incompatible with the lock, and the file cannot be truncated.

F_GETLK — Get the first lock which blocks the lock description given by the variable of type *struct flock* (see below) pointed to by *arg*. The information retrieved overwrites the information passed to *fcntl* in the structure *flock*. If no lock is found that would prevent this lock from being created, the structure is passed back unchanged except for the lock type which will be set to F_UNLCK.

F_SETLK — Set or clear a file segment lock according to the variable of type *struct flock* (see below) pointed to by *arg*. F_SETLK is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as remove either type of lock (F_UNLCK). F_RDLCK, F_WRLCK, and F_UNLCK are defined by the **<fcntl.h>** header file. If a read or write lock cannot be set, *fcntl* will return immediately with a return value of −1.

F_SETLKW — This command is the same as F_SETLK except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free to be locked.

The structure *flock* defined by the <**fcntl.h**> header file describes a lock. It describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*), RFS system ID, and process ID (*l_pid*) of the lock. The structure contains the following fields:

```
short   l_type;      /* F_RDLCK, F_WRLCK, or F_UNLCK */
short   l_whence;    /* SEEK_SET, SEEK_CUR, SEEK_END */
long    l_start;     /* Relative offset in bytes */
long    l_len;       /* Length, if 0 then until EOF */
short   l_sysid;     /* RFS system ID of process owning lock,
                        returned with F_GETLK */
short   l_pid;       /* Process ID of process owning lock,
                        returned with F_GETLK */
```

When a read lock has been set on a segment of a file, other processes may also set read locks on that segment or a portion of it; and even if enforcement-mode record locking is in effect, other processes may read the locked segment. A read lock prevents any other process from setting a write lock on any portion of the protected area; and if enforcement-mode record locking is in effect, other processes may not write to any portion of the protected area. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any other process from setting a read lock or a write lock on any portion of the protected area; and if enforcement-mode record locking is in effect, other processes may neither read nor write any portion of the protected area. The file descriptor on which a write lock is being placed must have been opened with write access.

The value of *l_whence* is SEEK_SET, SEEK_CUR, or SEEK_END (0, 1, or 2, respectively) to indicate that the relative offset *l_start* will be measured from the start of the file, the current position, or the end of the file, respectively. These symbolic values are defined in the <**unistd.h**> header file.

The value of *l_len* is the number of consecutive bytes to be locked. The process ID *l_pid* field is only used with F_GETLK to return the value for a blocking lock.

Locks may start and extend beyond the end of a file, but may not be negative relative to the beginning of the file. A lock may be set always to extend to the end of file by setting *l_len* to zero. If such a lock also has *l_start* set to zero and *l_whence* set to SEEK_SET, the whole file will be locked.

Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments locked at each end of the originally locked segment. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect.

All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process after executing the *fork*(2) routine.

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

[EAGAIN] *cmd* is F_SETLK the type of lock (*l_type*) is a read (F_RDLCK) lock and the segment of a file to be locked is already write locked by another process or the type is a write (F_WRLCK) lock and the segment of a file to be locked is already read or write locked by another process.

[EBADF] The *fildes* argument is not a valid open file descriptor, or the argument *cmd* is F_SETLK or F_SETLKW, the type of lock, *l_type*, is a shared lock (F_RDLCK), and *fildes* is not a valid file descriptor open for reading, or the type of lock *l_type*, is an exclusive lock (F_WRLCK), and *files* is not a valid file descriptor open for writing.

[EDEADLK]	*cmd* is F_SETLKW, the lock is blocked by some lock from another process, and putting the calling-process to sleep, waiting for that lock to become free, would cause a deadlock.
[EFAULT]	*cmd* is F_SETLK, *arg* points outside the program address space.
[EINTR]	A signal was caught during the *fcntl* system call.
[EINVAL]	*cmd* is F_DUPFD. *arg* is either negative, or greater than or equal to OPEN_MAX.
[EINVAL]	*cmd* is F_GETLK, F_SETLK, or SETLKW and *arg* or the data it points to is not valid.
[ENOLCK]	*cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock, and there are no more record locks available (too many file segments locked) because the system maximum has been exceeded.
[ENOLINK]	*fildes* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

close(2), creat(2), dup(2), exec(2), fork(2), open(2), pipe(2), fcntl(5).

DIAGNOSTICS

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_SETFD	Value other than −1.
F_GETFL	Value of file flags.
F_SETFL	Value other than −1.
F_GETLK	Value other than −1.

F_SETLK	Value other than −1.
F_SETLKW	Value other than −1.

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

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS

#include <stdio.h>

int ferror (stream)
FILE *stream;

int feof (stream)
FILE *stream;

void clearerr (stream)
FILE *stream;

int fileno (stream)
FILE *stream;

DESCRIPTION

ferror returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*, otherwise zero.

feof returns non-zero when **EOF** has previously been detected reading the named input *stream*, otherwise zero.

clearerr resets the error indicator and **EOF** indicator to zero on the named *stream*.

fileno returns the integer file descriptor associated with the named *stream*; see *open*(2).

NOTES

All these functions are implemented as macros; they cannot be declared or redeclared.

SEE ALSO

open(2), fopen(3S), stdio(3S).

DIAGNOSTICS

ferror, *fileno* and *feof* fails if:

[EBADF] The file descriptor underlying *stream* is not valid.

This page is intentionally left blank

NAME

floor, ceil, fmod, fabs – floor, ceiling, remainder, absolute value functions

SYNOPSIS

```
#include <math.h>

double floor (x)
  double x;

double ceil (x)
  double x;

double fmod (x, y)
  double x, y;

double fabs (x)
  double x;
```

DESCRIPTION

floor returns the largest integer (as a double-precision number) not greater than x.

ceil returns the smallest integer not less than x.

fmod returns the floating-point remainder of the division of x by y: x if y is zero or if x/y would overflow; otherwise the number f with the same sign as x, such that $x = iy + f$ for some integer i, and $|f| < |y|$.

fabs returns the absolute value of x, $|x|$.

If x (or y) is NaN, NaN is returned and *errno* is set to EDOM

SEE ALSO

abs(3C).

DIAGNOSTICS

The routines will fail if:

[EDOM]	x (or y) is NaN.
[ERANGE]	The result would overflow.

This page is intentionally left blank

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)
 int fildes;
 char *type;
```

DESCRIPTION

fopen opens the file named by *filename* and associates a *stream* with it. *fopen* returns a pointer to the FILE structure associated with the *stream*.

filename points to a character string that contains the name of the file to be opened.

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

"r"	open for reading
"w"	truncate or create for writing
"a"	append; open for writing at end of file, or create for writing
"r+"	open for update (reading and writing)
"w+"	truncate or create for update
"a+"	append; open or create for update at end-of-file

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

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

fdopen associates a *stream* with a file descriptor. File descriptors are obtained from *open*, *dup*, *creat*, or *pipe*(2), which open files but do not return pointers to a FILE structure *stream*. Streams are necessary input for many of the Section 3S library routines. The *type* of *stream* must agree with the mode of the open file.

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

When a file is opened for append (i.e., when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. *fseek* may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

If *mode* is "w", "a", "w+" or "a+" and the file did not previously exist, upon successful completion the *fopen*(2) function will mark for update the *st_atime*, *st_ctime* and *st_mtime* fields of the file and the *st_ctime* and *st_mtime* fields of the parent directory.

If *mode* is "w" or "w+" and the file did previously exist, upon successful completion the *fopen*(2) function will mark for update the *st_ctime* and *st_mtime* fields of the file. The *fopen*(2) function will allocate a file descriptor as *open*(2) does.

SEE ALSO

creat(2), dup(2), open(2), pipe(2), fclose(3S), fseek(3S), stdio(3S).

DIAGNOSTICS

fopen, *fdopen*, and *freopen* return a NULL pointer on failure.

<u>The *fdopen*() function may fail if:</u>

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The *mode* argument is not a valid mode.
[ENOMEM]	Insufficient space to allocate a buffer.

<u>The *fopen*() (*freopen*()) will fail if:</u>

[EACCES]	Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *mode* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.
[EINTR]	A signal was caught during the *fopen*() (*freopen*()) function.
[EISDIR]	The named file is a directory and *mode* requires write access.
[EMFILE]	FOPEN_MAX file descriptors, directories and message catalogues are currently open in the calling process.
[ENAMETOOLONG]	The length of the *filename* string exceeds PATH_MAX or a pathname component is longer than NAME_MAX while _POSIX_NO_TRUNC is in effect.
[ENFILE]	The system file table is full.
[ENOENT]	The named file does not exist or the *filename* argument points to an empty string.

[ENOSPC]	The directory or file system that would contain the new file cannot be expanded, the file does not exist, and it was to be created.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
[EROFS]	The named file resides on a read-only file system and *mode* requires write access.

The *fopen*() function may fail if:

[EINVAL]	The value of the *mode* argument is not valid.
[ENOMEM]	Insufficient storage space is available.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and *mode* requires write access.

NAME

fork – create a new process

SYNOPSIS

#include <sys/types.h>

pid_t fork ()

DESCRIPTION

fork causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

environment
close-on-exec flag [see *exec*(2)]
signal handling settings (i.e.,**SIG_DFL, SIG_IGN, SIG_HOLD**, function address)
set-user-ID mode bit
set-group-ID mode bit
profiling on/off status
nice value [see *nice*(2)]
all attached shared memory segments [see *shmop*(2)]
process group ID
tty group ID [see *exit*(2)]
current working directory
root directory
file mode creation mask [see *umask*(2)]
file size limit [see *ulimit*(2)]
MCU mask [see *mcumask*(2)]

The child process differs from the parent process in the following ways:

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 file descriptors. Each of the child's file descriptors shares a a common file pointer with the corresponding file descriptor of the parent.

All semadj values are cleared [see *semop*(2)].

Process locks, text locks and data locks are not inherited by the child [see *plock*(2)].

The child process's *utime, stime, cutime*, and *cstime* are set to 0. The time left until an alarm clock signal is reset to 0.

The child process will be running on the same MCU as the parent process.

There is a mechanism that enables the calling process to control the position of the new process in the process hierarchy. Normally, the new process becomes a child of the calling process. If, however, the *fork* call is preceded by the following call:

set_parm(biology, −1, −1);

the situation changes. The argument *biology* controls the relationship between the new process and the calling process. If *biology* is 0, the situation is the same as above: The new process becomes a child of the calling process. If *biology* is 1, the new process becomes a child of process number 1. If *biology* is 2, the new process becomes a sibling of the calling process, that is, the parent process ID of the new process is set to the parent process ID of the calling process.

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

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

[ENOMEM] The process requires more space than the system is able to supply.

SEE ALSO

exec(2), nice(2), plock(2), ptrace(2), semop(2), shmop(2), signal(2), sigset(2), times(2), ulimit(2), umask(2), wait(2).

DIAGNOSTICS

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 *errno* is set to indicate the error.

This page is intentionally left blank

NAME

fpathconf, pathconf – get configurable pathname variables

SYNOPSIS

#include <unistd.h>

long fpathconf (fildes, name)
  int fildes, name;

long pathconf (path, name)
  char *path;
  int name;

DESCRIPTION

The functions *fpathconf* and *pathconf* return the current value of a configurable limit or option associated with a file or directory. The *path* argument points to the pathname of a file or directory; *fildes* is an open file descriptor; and *name* is the symbolic constant (defined in <**unistd.h**>) representing the configurable system limit or option to be returned.

Value of *name*	See Note
_PC_LINK_MAX	1
_PC_MAX_CANNON	2
_PC_MAX_INPUT	2
_PC_NAME_MAX	3,4
_PC_PATH_MAX	4,5
_PC_PIPE_BUF	6
_PC_CHOWN_RESTRICTED	7
_PC_NO_TRUNC	3,4
_PC_VDISABLE	2

The values returned by *pathconf* and *fpathconf* depend on the type of file specified by *path* or *fildes*. The table contains the symbolic constants supported by *pathconf* and *fpathconf* along with the POSIX defined return value. The return value is based on the type of file specified by *path* or *fildes*.

Notes:

1 If *path* or *fildes* refers to a directory, the value returned applies to the directory itself.

2 The behavior is undefined if *path* or *fildes* does not refer to a terminal file.

3 If *path* or *fildes* refers to a directory, the value returned applies to the filenames within the directory.

4 The behavior is undefined if *path* or *fildes* does not refer to a directory.

5 If *path* or *fildes* refers to a directory, the value returned is the maximum length of a relative pathname when the specified directory is the working directory.

6 If *path* or *fildes* refers to a pipe or FIFO, the value returned applies to the FIFO itself. If *path* or *fildes* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If *path* or *fildes* refer to any other type of file, the behavior is undefined.

7 If *path* or *fildes* refers to a directory, the value returned applies to any files, other than directories, that exist or can be created within the directory.

The value of the configurable system limit or option specified by *name* does not change during the lifetime of the calling process.

fpathconf fails if the following is true:

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

pathconf fails if one or more of the following are true:

[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links are encountered while translating *path*.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines and file system type does not allow it.
[ENAMETOOLONG]	The length of a pathname exceeds {PATH_MAX}, or pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOENT]	*path* is needed for the command specified and the named file does not exist or if the *path* argument points to an empty string.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[ENOTDIR]	a component of the path prefix is not a directory.

Both *fpathconf* and *pathconf* fail if the following is true:

[EINVAL]	if *name* is an invalid value.

SEE ALSO

sysconf(3C), limits(4).

DIAGNOSTICS

If *fpathconf* or *pathconf* are invoked with an invalid symbolic constant or the symbolic constant corresponds to a configurable system limit or option not supported on the system, a value of −1 is returned to the invoking process.

If the function fails because the configurable system limit or option corresponding to *name* is not supported on the system the value of *errno* is not changed.

NAME

fread, fwrite – binary input/output

SYNOPSIS

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

size_t fread (ptr, size, nitems, stream)
 void *ptr;
 size_t size, nitems;
 FILE *stream;

size_t fwrite (ptr, size, nitems, stream)
 void *ptr;
 size_t size, nitems;
 FILE *stream;
```

DESCRIPTION

fread copies, into an array pointed to by *ptr*, *nitems* items of data from the named input *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. *fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. *fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. *fread* does not change the contents of *stream*.

fwrite appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream*. *fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. *fwrite* does not change the contents of the array pointed to by *ptr*.

The argument *size* is typically *sizeof(*ptr)* where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than *char* it should be cast into a pointer to *char*.

SEE ALSO

read(2), write(2), fopen(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S), stdio(3S).

DIAGNOSTICS

fread and *fwrite* return the number of items read or written. If *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

NAME

frexp, ldexp, modf – manipulate parts of floating-point numbers

SYNOPSIS

```
#include <math.h>

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

double ldexp (value, exp)
 double value;
 int exp;

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

DESCRIPTION

Every non-zero number can be written uniquely as $x * 2^n$, where the "mantissa" (fraction) x is in the range $0.5 \leq |x| < 1.0$, and the "exponent" n is an integer. *frexp* returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*. If *value* is zero, both results returned by *frexp* are zero.

ldexp returns the quantity $value * 2^{exp}$.

modf returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

DIAGNOSTICS

If *ldexp* would cause overflow, ±**HUGE_VAL** (defined in **<math.h>**) is returned (according to the sign of *value*), and *errno* is set to ERANGE.

If *ldexp* would cause underflow, zero is returned and *errno* is set to ERANGE.

If these functions are called with a value equal to NaN, NaN is returned and *errno* is set to EDOM.

This page is intentionally left blank

NAME

fseek, rewind, ftell – reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>

int fseek (stream, offset, ptrname)
 FILE *stream;
 long offset;
 int ptrname;

void rewind (stream)
 FILE *stream;

long ftell (stream)
 FILE *stream;
```

DESCRIPTION

fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, according as *ptrname* has the value SEEK_SET, SEEK_CUR, or SEEK_END.

rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, SEEK_SET), except that no value is returned and the error indicator is cleared.

fseek and *rewind* undo any effects of *ungetc*(3S).

After *fseek* or *rewind*, the next operation on a file opened for update may be either input or output.

ftell returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

SEE ALSO

lseek(2), fopen(3S), popen(3S), stdio(3S), ungetc(3S).

DIAGNOSTICS

fseek returns non-zero for improper seeks, otherwise zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen*; in particular, *fseek* may not be used on a terminal, or on a file opened via *popen*(3S).

WARNING

Although on the UNIX system an offset returned by *ftell* is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non-UNIX systems requires that an offset be used by *fseek* directly. Arithmetic may not meaningfully be performed on such an offset, which is not necessarily measured in bytes.

NAME

fsync – synchronize a file's in-memory state with that on the physical medium

SYNOPSIS

#include <unistd.h>

int fsync(fildes)
int fildes;

DESCRIPTION

fsync moves all modified data and attributes of *fildes* to a storage device. When *fsync* returns, all in-memory modified copies of buffers associated with *fildes* have been written to the physical medium. *fsync* is different from *sync*, which schedules disk I/O for all files but returns before the I/O completes.

fsync should be used by programs that require that a file be in a known state. For example, a program that contains a simple transaction facility might use *fsync* to ensure that all changes to a file or files caused by a given transaction were recorded on a storage medium.

fsync fails if one or more of the following are true:

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

[EINTR] A signal was caught during execution of the `fsync` system call.

[EIO] An I/O error occurred while reading from or writing to the file system.

[ENOLINK] *fildes* is on a remote machine and the link on that machine is no longer active.

DIAGNOSTICS

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

NOTES

The way the data reach the physical medium depends on both implementation and hardware. *fsync* returns when the device driver tells it that the write has taken place.

SEE ALSO

sync(2)

NAME

ftw – walk a file tree

SYNOPSIS

#include <ftw.h>

```
int ftw (path, fn, depth)
 char *path;
 int (*fn) ( );
 int depth;
```

DESCRIPTION

ftw recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, *ftw* calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure [see *stat*(2)] containing information about the object, and an integer. Possible values of the integer, defined in the <ftw.h> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which *stat* could not successfully be executed. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the **stat** structure will contain garbage. An example of an object that would cause FTW_NS to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

ftw visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or some error is detected within *ftw* (such as an I/O error). If the tree is exhausted, *ftw* returns zero. If *fn* returns a nonzero value, *ftw* stops its tree traversal and returns whatever value was returned by *fn*. If *ftw* detects an error, it returns −1, and sets the error type in *errno*.

ftw uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. If *depth* is zero or negative, the effect is the same as if it were 1. *Depth* must not be greater than the number of file descriptors currently available for use. *ftw* will run more quickly if *depth* is at least as large as the number of levels in the tree.

SEE ALSO

stat(2), malloc(3C).

BUGS

Because *ftw* is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.

CAVEAT

ftw uses *malloc*(3C) to allocate dynamic storage during its operation. If *ftw* is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, *ftw* will not have a chance to free that storage, so it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

DIAGNOSTICS

The *ftw*( ) function will fail if:

[EACCES]	Search permission is denied for any component of *path* or read permission is denied for *path*.
[EINVAL]	The value of the *ndirs* argument is invalid.
[ENAMETOOLONG]	The length of the *path* string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOENT]	The *path* argument points to the name of a file which does not exist or points to an empty string.
[ENOTDIR]	A component of *path* is not a directory.

NAME

gamma – log gamma function

SYNOPSIS

#include <math.h>

double gamma (x)
double x;

double lgamma (x)
double x;

extern int signgam;

DESCRIPTION

gamma and *lgamma* returns $\ln(|\Gamma(x)|)$, where $\Gamma(x)$ is defined as $\int_0^\infty e^{-t} t^{x-1} dt$. The sign of $\Gamma(x)$ is returned in the external integer *signgam*. The argument x may not be a non-positive integer.

The following C program fragment might be used to calculate Γ:

```
if ((y = gamma(x)) > LN_MAXDOUBLE)
        error( );
y = signgam * exp(y);
```

where LN_MAXDOUBLE is the least value that causes *exp*(3M) to return a range error, and is defined in the *<values.h>* header file.

SEE ALSO

exp(3M), values(5).

DIAGNOSTICS

For non-negative integer arguments **HUGE_VAL** is returned, and *errno* is set to **EDOM**. A message indicating SING error is printed on the standard error output.

If the correct value would overflow, *gamma* returns **HUGE_VAL** and sets *errno* to **ERANGE**.

If x is NaN, NaN is returned and *errno* is set to EDOM.

This page is intentionally left blank

NAME

getc, getchar, fgetc, getw – get character or word from a 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 (i.e., byte) from the named input *stream*, as an integer. It also moves the file pointer, if defined, ahead one character in *stream*. *getchar* is defined as *getc(stdin)*. *getc* and *getchar* are macros.

fgetc behaves like *getc*, but is a function rather than a macro. *fgetc* runs more slowly than *getc*, but it takes less space per invocation and its name can be passed as an argument to a function.

getw returns the next word (i.e., integer) from the named input *stream*. *getw* increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. *getw* assumes no special alignment in the file.

The functions may mark the *st_atime* fields of the file associated with *stream* for update. The *st_atime* field will be marked for update by the first successful execution of *fgetc*( ), *fgets*( ), *fread*( ), *getc*( ), *getchar*( ), *gets*( ) or *fscanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ).

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S), stdio(3S).

DIAGNOSTICS

These functions return the constant **EOF** at end-of-file or upon an error. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *getw* errors.

The functions will fail if:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlaying *stream* and the process would be delayed in the *fgetc*( ) operation.

[EBADF] The file descriptor underlaying *stream* is not a valid file descriptor open for reading.

[EINTR] The read operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.

[EIO] The implementation supports job control, the process is a member of a background process attempting to read from its controlling terminal, the process is either ignoring or blocking the SIGTTIN signal or the process group is orphaned. This error may also be generated for implementation-defined reasons.

[ENOMEM] Insufficient storage space is available.

[ENXIO] A request was made of a non-existent device, or the request was outside the capabilities of the device.

WARNING

If the integer value returned by *getc*, *getchar*, or *fgetc* is stored into a character variable and then compared against the integer constant **EOF**, the comparison may never succeed, because sign-extension of a character on widening to integer is machine-dependent.

CAVEATS

Because it is implemented as a macro, *getc* evaluates a *stream* argument more than once. In particular, **getc(*f++)** does not work sensibly. *fgetc* should be used instead.

Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

This page is intentionally left blank

NAME

getcwd – get path-name of current working directory

SYNOPSIS

char * getcwd (buf, size)
char * buf;
int size;

DESCRIPTION

getcwd returns a pointer to the current directory path name. The value of *size* must be at least two greater than the length of the path-name to be returned.

If *buf* is a NULL pointer, *getcwd* will obtain *size* bytes of space using *malloc*(3C). In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free.*

The function is implemented by using *popen*(3S) to pipe the output of the *pwd*(1) command into the specified string space.

EXAMPLE

```
void exit(), perror();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
    perror("pwd");
    exit(2);
}
printf("%s\n", cwd);
```

SEE ALSO

malloc(3C), popen(3S), pwd(1).

DIAGNOSTICS

Returns **NULL** with *errno* set if *size* is not large enough, or if an error occurs in a lower-level function.

The *getcwd*( ) function will fail if:

[EACCES]	Read or search permission was denied for a component of the pathname.
[EINVAL]	The *size* argument is zero or negative.
[ENOMEM]	Insufficient storage space is available.
[ERANGE]	The size argument is greater than zero, but is smaller than the length of the pathname + 1.

NAME

getdents – read directory entries and put in a file system independent format

SYNOPSIS

#include <sys/dirent.h>

```
int getdents (fildes, buf, nbyte)
 int fildes;
 char *buf;
 unsigned nbyte;
```

DESCRIPTION

fildes is a file descriptor obtained from an *open*(2) or *dup*(2) system call.

getdents attempts to read *nbyte* bytes from the directory associated with *fildes* and to format them as file system independent directory entries in the buffer pointed to by *buf*. Since the file system independent directory entries are of variable length, in most cases the actual number of bytes returned will be strictly less than *nbyte*.

The file system independent directory entry is specified by the *dirent* structure. For a description of this see *dirent*(4).

On devices capable of seeking, *getdents* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *getdents,* the file pointer is incremented to point to the next directory entry.

This system call was developed in order to implement the *readdir*(3X) routine [for a description see *directory*(3C)], and should not be used for other purposes.

getdents 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.

[EINVAL] *nbyte* is not large enough for one directory entry.

[EIO] An I/O error occurred while accessing the file system.

[ENOENT] The current file pointer for the directory is not located at a valid entry.

[ENOLINK] *fildes* points to a remote machine and the link to that machine is no longer active.

[ENOTDIR] *fildes* is not a directory.

SEE ALSO

directory(3C), dirent(4).

DIAGNOSTICS

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. A value of 0 indicates the end of the directory has been reached. If the system call failed, a -1 is returned and *errno* is set to indicate the error.

NAME

getenv – return value for environment name

SYNOPSIS

#include <stdlib.h>

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 *value* in the current environment if such a string is present, otherwise a NULL pointer.

SEE ALSO

exec(2), putenv(3C) and environ(5).

This page is intentionally left blank

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent, fgetgrent – get group file entry

SYNOPSIS

#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (gid)
 gid_t gid;

struct group *getgrnam (name)
 char *name;

void setgrent ( )

void endgrent ( )

struct group *fgetgrent (f)
 FILE *f;

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 **/etc/group** file. Each line contains a "group" structure, defined in the *<grp.h>* header file.

```
struct              group {
  char *gr_name;    /* the name of the group */
  char *gr_passwd;  /* the encrypted group password */
  int gr_gid;       /* the numerical group ID */
  char **gr_mem;    /* vector of pointers to member names */
};
```

getgrent when first called returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls may be used to search the entire file. *getgrgid* searches from the beginning of the file until a numerical group id matching *gid* is found and returns a pointer to the particular structure in which it was found. *getgrnam* searches from the beginning of the file until a group name matching *name* is found and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. *endgrent* may be called to close the group file when processing is complete.

fgetgrent returns a pointer to the next group structure in the stream *f*, which matches the format of **/etc/group**.

FILES

/etc/group

SEE ALSO

getlogin(3C), getpwent(3C), group(4).

DIAGNOSTICS

A **NULL** pointer is returned on **EOF** or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

CAVEAT

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getgroups – get supplementary group access list IDs

SYNOPSIS

#include <unistd.h>

int getgroups(gidsetsize, grouplist)
int gidgetsize;
gid_t grouplist[ ];

DESCRIPTION

getgroups gets the current supplemental group access list of the calling process and stores the result in the array of group IDs specified by *grouplist*. This array has *gidsetsize* entries and must be large enough to contain the entire list. This list cannot be greater than {NGROUPS_MAX}. If *gidsetsize* equals 0, *getgroups* will return the number of groups to which the calling process belongs without modifying the array pointed to by *grouplist*.

getgroups will fail if:

[EFAULT] A referenced part of the array pointed to by *grouplist* is outside of the allocated address space of the process.

[EINVAL] The value of *gidsetsize* is non-zero and less than the number of supplementary group IDs set for the calling process.

SEE ALSO

chown(2), getuid(2), setuid(2), initgroups(3C).

DIAGNOSTICS

Upon successful completion, *getgroups* returns the number of supplementary group IDs set for the calling process and *setgroups* returns the value 0. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

This page is intentionally left blank

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 user ID is shared by several login names.

If *getlogin* is called within a process that is not attached to a terminal, it returns a **NULL** pointer. The correct procedure for determining the login name is to call *cuserid*, or to call *getlogin* and if it fails to call *getpwuid*.

FILES

/etc/utmp

SEE ALSO

cuserid(3S), getgrent(3C), getpwent(3C), utmp(4).

DIAGNOSTICS

Returns the **NULL** pointer if *name* is not found.

CAVEAT

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

This page is intentionally left blank

NAME

getmsg – get next message off a stream

SYNOPSIS

#include <stropts.h>

```
int getmsg(fd, ctlptr, dataptr, flags)
 int fd;
 struct strbuf *ctlptr;
 struct strbuf *dataptr;
 int *flags;
```

DESCRIPTION

getmsg retrieves the contents of a message [see *intro*(2)] located at the *stream head* read queue from a STREAMS file, and places the contents into user specified buffer(s). The message must contain either a data part, a control part or both. The data and control parts of the message are placed into separate buffers, as described below. The semantics of each part is defined by the STREAMS module that generated the message.

fd specifies a file descriptor referencing an open *stream*. *ctlptr* and *dataptr* each point to a *strbuf* structure which contains the following members:

```
int maxlen;  /* maximum buffer length */
int len;     /* length of data        */
char *buf;   /* ptr to buffer    */
```

where *buf* points to a buffer in which the data or control information is to be placed, and *maxlen* indicates the maximum number of bytes this buffer can hold. On return, *len* contains the number of bytes of data or control information actually received, or is 0 if there is a zero-length control or data part, or is −1 if no data or control information is present in the message. *flags* may be set to the values 0 or RS_HIPRI and is used as described below.

ctlptr is used to hold the control part from the message and *dataptr* is used to hold the data part from the message. If *ctlptr* (or *dataptr*) is NULL or the *maxlen* field is −1, the control (or data) part of the message is not processed and is left on the *stream head* read queue and *len* is set to −1. If the *maxlen* field is set to 0 and there is a zero-length control (or data) part, that zero-length part is removed from the read queue and *len* is set to 0. If the *maxlen* field is set to 0 and there are more than zero bytes of control (or data) information, that information is left on the read queue and *len* is set to 0. If the *maxlen* field in *ctlptr* or *dataptr* is less than, respectively, the control or data part of the message, *maxlen* bytes are retrieved. In this case, the remainder of the message is left on the *stream head* read queue and a non-zero return value is provided, as described below under *DIAGNOSTICS*. If information is retrieved from a *priority* message, *flags* is set to RS_HIPRI on return.

By default, *getmsg* processes the first priority or non-priority message available on the *stream head* read queue. However, a user may choose to retrieve only priority messages by setting *flags* to RS_HIPRI. In this case, *getmsg* will only process the next message if it is a priority message.

If O_NDELAY has not been set, *getmsg* blocks until a message, of the type(s) specified by *flags* (priority or either), is available on the *stream head* read queue. If O_NDELAY has been set and a message of the specified type(s) is not present on the read queue, *getmsg* fails and sets *errno* to EAGAIN.

If a hangup occurs on the *stream* from which messages are to be retrieved, *getmsg* will continue to operate normally, as described above, until the *stream head* read queue is empty. Thereafter, it will return 0 in the *len* fields of *ctlptr* and *dataptr*.

getmsg fails if one or more of the following are true:

[EAGAIN] The O_NDELAY flag is set, and no messages are available.

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

[EBADMSG] Queued message to be read is not valid for *getmsg*.

[EFAULT] *ctlptr*, *dataptr*, or *flags* points to a location outside the allocated address space.

[EINTR] A signal was caught during the *getmsg* system call.

[EINVAL] An illegal value was specified in *flags*, or the *stream* referenced by *fd* is linked under a multiplexor.

[ENOSTR] A *stream* is not associated with *fd*.

A *getmsg* can also fail if a STREAMS error message had been received at the *stream head* before the call to *getmsg*.

SEE ALSO

intro(2), poll(2), putmsg(2), read(2), write(2).

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. A value of 0 indicates that a full message was read successfully. A return value of MORECTL indicates that more control information is waiting for retrieval. A return value of MOREDATA indicates that more data is waiting for retrieval. A return value of MORECTL | MOREDATA indicates that both types of information remain. Subsequent *getmsg* calls will retrieve the remainder of the message.

This page is intentionally left blank

NAME

getopt – get option letter from argument vector

SYNOPSIS

```
#include <stdio.h>

int getopt (argc, argv, optstring)
 int argc;
 char * * argv, * opstring;

extern char * optarg;
extern int optind, opterr;
```

DESCRIPTION

getopt returns the next option letter in *argv* that matches a letter in *optstring*. It supports all the rules of the command syntax standard (see *intro*(1)). So all new commands will adhere to the command syntax standard, they should use *getopts*(1) or *getopt*(3C) to parse positional parameters and check for options that are legal for that command.

optstring must contain the option letters the command using *getopt* will recognize; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must 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. **optind** is external and is initialized to **1** before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns **–1**. The special option "– –" may be used to delimit the end of the options; when it is encountered, **–1** will be returned, and "– –" will be skipped.

DIAGNOSTICS

getopt prints an error message on standard error and returns a question mark (**?**) when it encounters an option letter not included in *optstring* or no option-argument after an option that expects one. This error message may be disabled by setting **opterr** to **0**.

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 option **o**, which requires an option-argument:

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

main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern char *optarg;
        extern int optind;
        .
        .
        .
        while ((c = getopt(argc, argv, "abo:")) != EOF)
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
                                bproc( );
                        break;
                case 'o':
                        ofile = optarg;
                        break;
                case '?':
```

```
                errflg++;
            }
        if (errflg) {
            (void)fprintf(stderr, "usage: . . . ");
            exit (2);
        }
        for ( ; optind < argc; optind++) {
            if (access(argv[optind], R_OK)) {
        .
        .
        .
}
```

WARNING

Although the following command syntax rule (see *intro*(1)) relaxations are permitted under the current implementation, they should not be used because they may not be supported in future releases of the system. As in the **EXAMPLE** section above, **a** and **b** are options, and the option **o** requires an option-argument:

cmd −aboxxx file
(Rule 5 violation: options with option-arguments must not be grouped with other options)

cmd −ab −oxxx file
(Rule 6 violation: there must be white space after an option that takes an option-argument)

SEE ALSO

getopts(1), intro(1).

This page is intentionally left blank

NAME

getpass – read a password

SYNOPSIS

char *getpass (prompt)
char *prompt;

DESCRIPTION

getpass reads up to a newline or **EOF** from the file **/dev/tty**, after prompting on the standard error output with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most PASS_MAX characters. If **/dev/tty** cannot be opened, a **NULL** pointer is returned. An interrupt will terminate input and send an interrupt signal to the calling program before returning.

The *getpass*() function marks for update the *st_atime* and *st_mtime* fields of the file **/dev/tty**.

FILES

/dev/tty

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs not otherwise using standard I/O, more than might be expected.

CAVEAT

The return value points to static data whose content is overwritten by each call.

This page is intentionally left blank

NAME

getpid, getpgrp, getppid – get process, process group, and parent process IDs

SYNOPSIS

#include <sys/types.h>

pid_t getpid ()

pid_t getpgrp ()

pid_t getppid ()

DESCRIPTION

getpid returns the process ID of the calling process.

getpgrp returns the process group ID of the calling process.

getppid returns the parent process ID of the calling process.

SEE ALSO

exec(2), fork(2), intro(2), setpgrp(2), signal(2).

This page is intentionally left blank

NAME

getpw – get name from UID

SYNOPSIS

```
int getpw (uid, buf)
 int uid;
 char *buf;
```

DESCRIPTION

getpw searches the password file for a user id number that equals *uid*, copies the line of the password file in which *uid* was found into the array pointed to by *buf*, and returns 0. *getpw* returns non-zero if *uid* cannot be found.

This routine is included only for compatibility with prior systems and should not be used; see *getpwent*(3C) for routines to use instead.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(4).

DIAGNOSTICS

getpw returns non-zero on error.

WARNING

The above routine uses **<stdio.h>**, which causes it to increase, more than might be expected, the size of programs not otherwise using standard I/O.

This page is intentionally left blank

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent, fgetpwent – get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent ()

struct passwd *getpwuid (uid)
 uid_t uid;

struct passwd *getpwnam (name)
 char *name;

void setpwent ()

void endpwent ()

struct passwd *fgetpwent (f)
 FILE *f;
```

DESCRIPTION

getpwent, *getpwuid* and *getpwnam* each returns a pointer to an object with the following structure containing the broken-out fields of a line in the **/etc/passwd** file. Each line in the file contains a "passwd" structure, declared in the *<pwd.h>* header file:

```
struct passwd {
        char    *pw_name;
        char    *pw_passwd;
        uid_t   pw_uid;
        gid_t   pw_gid;
        char    *pw_age;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};
```

This structure is declared in <*pwd.h*> so it is not necessary to redeclare it.

The fields have meanings described in *passwd*(4).

getpwent when first called returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; so successive calls can be used to search the entire file. *getpwuid* searches from the beginning of the file until a numerical user id matching *uid* is found and returns a pointer to the particular structure in which it was found. *getpwnam* searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *endpwent* may be called to close the password file when processing is complete.

fgetpwent returns a pointer to the next passwd structure in the stream *f*, which matches the format of **/etc/passwd**.

FILES

/etc/passwd

SEE ALSO

getgrent(3C), getlogin(3C), passwd(4).

DIAGNOSTICS

A **NULL** pointer is returned on **EOF** or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

CAVEAT

All information is contained in a static area, so it must be copied if it is to be saved.

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;
 int n;
 FILE *stream;
```

DESCRIPTION

gets reads characters from the standard input stream, *stdin,* into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

fgets reads characters from the *stream* into the array pointed to by *s*, until $n-1$ characters are read, or a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

The *gets*( ) and *fgets*( ) functions may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field will be marked update by the first successful execution of *fgetc*( ), *fgets*( ), *fread*( ), *getc*( ), *getchar*( ), *gets*( ) or *fscanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ).

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), scanf(3S), stdio(3S).

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

gets and *fgets* will fail if:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlaying *stream* and the process would be delayed in the *gets*( ) operation.

[EBADF] The file descriptor underlaying *stream* is not a valid file descriptor open for reading.

[EINTR] The read operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.

[EIO] The implementation supports job control, the process is a memeber of a background process attempting to read from its controlling terminal, the process is either ignoring or blocking the SIGTTIN signal or the process group is orphaned. This error may also be generated for implementation-defined reasons.

[ENOMEM] Insufficient storage space is available.

[ENXIO] A request was made of a non-existent device, or the request was outside the capabilities of the device.

NAME

gettxt – retrieve a text string

SYNOPSIS

#include <nl_types.h>

char * gettxt (const char * msgid, const char * dflt_str);

DESCRIPTION

gettxt retrieves a text string from a message file. The arguments to the function are a message identification *msgid* and a default string *dflt_str* to be used if the retrieval fails.

The text strings are in files created by the `mkmsgs` utility (see *mkmsgs*(1)) and installed in directories in

/usr/lib/locale/<*locale*>**/LC_MESSAGES**.

The directory <*locale*> can be viewed as the language in which the text strings are written. The user can request that messages be displayed in a specific language by setting the environment variable **LC_MESSAGES**. If **LC_MESSAGES** is not set the environment variable **LANG** will be used. If **LANG** is not set, the files containing the strings are in

/usr/lib/locale/C/LC_MESSAGES/ * .

The user can also change the language in which the messages are displayed in by invoking the *setlocale* function with the appropriate arguments.

If *gettxt* fails to retrieve a message in a specific language it will try to retrieve the same message in U.S. English. On failure, the processing depends on what the second argument *dflt_str* points to. A pointer to the second argument is returned if the second argument is not the null string. If *dflt_str* points to the null string a pointer to the U.S. English text string "`Message not found!!\n`" is returned.

The following depicts the acceptable syntax of *msgid* for a call to *gettxt*.

<msgid> = <*msgfilename*>:<*msgnumber*>

The first field is used to indicate the file that contains the text strings and must be limited to 14 characters. These characters must be selected from the set of all character values excluding \0 (null) and the ASCII code for / (slash) and : (colon). The names of message files must be the same as the names of files created by *mkmsgs* and installed in

/usr/lib/locale/<*locale*>**/LC_MESSAGES/** * .

The numeric field indicates the sequence number of the string in the file. The strings are numbered from **1** to **n** where *n* is the number of strings in the file.

On failure to pass the correct *msgid* or a valid message number to *gettxt* a pointer to the text string "`Message not found!!`\n" is returned.

EXAMPLE

gettxt("UX:10", "hello world\n")
gettxt("UX:10", "")

UX is the name of the file that contains the messages. **10** is the message number.

FILES

/usr/lib/locale/C/LC_MESSAGES/ *
: contains default message files created by *mkmsgs*.

/usr/lib/locale/*locale*/LC_MESSAGES/ *
: contains message files for different languages created by *mkmsgs*.

SEE ALSO

mkmsgs(1), setlocale(3C), environ(5) in the *System V Reference Manual*.

NAME

getuid, geteuid, getgid, getegid – get real user, effective user, real group, and effective group IDs

SYNOPSIS

#include <sys/types.h>

uid_t getuid ( )

uid_t geteuid ( )

gid_t getgid ( )

gid_t getegid ( )

DESCRIPTION

getuid returns the real user ID of the calling process.

geteuid returns the effective user ID of the calling process.

getgid returns the real group ID of the calling process.

getegid returns the effective group ID of the calling process.

SEE ALSO

intro(2), setuid(2).

This page is intentionally left blank

NAME

getut: getutent, getutid, getutline, pututline, setutent, endutent, utmpname − access utmp file entry

SYNOPSIS

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

struct utmp *getutent ()

struct utmp *getutid (id)
 struct utmp *id;

struct utmp *getutline (line)
 struct utmp *line;

void pututline (utmp)
 struct utmp *utmp;

void setutent ()

void endutent ()

void utmpname (file)
 char *file;
```

DESCRIPTION

getutent, *getutid* and *getutline* each return a pointer to a structure of the following type:

```
struct  utmp {
 char   ut_user[8];        /* User login name */
 char   ut_id[4];          /* /etc/inittab id (usually line #) */
 char   ut_line[12];       /* device name (console, lnxx) */
 short  ut_pid;            /* process id */
 short  ut_type;           /* type of entry */
 struct exit_status {
  short e_termination;     /* Process termination status */
  short e_exit;            /* Process exit status */
 } ut_exit;                /* The exit status of a process
                            * marked as DEAD_PROCESS. */
 time_t ut_time;           /* time entry was made */
};
```

getutent reads in the next entry from a *utmp*-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

getutid searches forward from the current point in the *utmp* file until it finds an entry with a *ut_type* matching *id->ut_type* if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME or NEW_TIME.

If the type specified in *id* is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS or DEAD_PROCESS, then *getutid* will return a pointer to the first entry whose type is one of these four and whose *ut_id* field matches *id->ut_id*. If the end of file is reached without a match, it fails.

getutline searches forward from the current point in the *utmp* file until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which also has a *ut_line* string matching the *line->ut_line* string. If the end of file is reached without a match, it fails.

pututline writes out the supplied *utmp* structure into the *utmp* file. It uses *getutid* to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of *pututline* will have searched for the proper entry using one of the *getut* routines. If so, *pututline* will not search. If *pututline* does not find a matching slot for the new entry, it will add a new entry to the end of the file.

setutent resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

endutent closes the currently open file.

utmpname allows the user to change the name of the file examined, from **/etc/utmp** to any other file. It is most often expected that this other file will be **/etc/wtmp**. If the file does not exist, this will not be apparent until the first attempt to reference the file is made. *utmpname* does not open the file. It just closes the old file if it is currently open and saves the new file name.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

DIAGNOSTICS

A NULL pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

NOTES

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either *getutid* or *getutline* sees the routine examine the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason to use *getutline* to search for multiple occurrences, it would be necessary to zero out the static after each success, or *getutline* would just return the same pointer over and over again. There is one exception to the rule about removing the structure before further reads are done. The implicit read done by *pututline* (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the *getutent*, *getutid* or *getutline* routines, if the user has just modified those contents and passed the pointer back to *pututline*.

These routines use buffered standard I/O for input, but *pututline* uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the *utmp* and *wtmp* files.

This page is intentionally left blank

NAME

hsearch, hcreate, hdestroy – manage hash search tables

SYNOPSIS

```
#include <search.h>

ENTRY * hsearch (item, action)
 ENTRY item;
 ACTION action;

int hcreate (nel)
 unsigned nel;

void hdestroy ( )
```

DESCRIPTION

hsearch is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. *item* is a structure of type ENTRY (defined in the <*search.h*> header file) containing two pointers: *item.key* points to the comparison key, and *item.data* points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) *action* is a member of an enumeration type ACTION indicating the disposition of the entry if it cannot be found in the table. **ENTER** indicates that the item should be inserted in the table at an appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

hcreate allocates sufficient space for the table, and must be called before *hsearch* is used. *nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

hdestroy destroys the search table, and may be followed by another call to *hcreate*.

NOTES

hsearch uses *open addressing* with a *multiplicative* hash function. However, its source code has many other options available which the user may select by compiling the *hsearch* source with the following symbols defined to the preprocessor:

DIV Use the *remainder modulo table size* as the hash function instead of the multiplicative algorithm.

USCR Use a User Supplied Comparison Routine for ascertaining table membership. The routine should be named *hcompar* and should behave in a mannner similar to *strcmp* [see *string*(3C)].

CHAINED
Use a linked list to resolve collisions. If this option is selected, the following other options become available.

START Place new entries at the beginning of the linked list (default is at the end).

SORTUP Keep the linked list sorted by key in ascending order.

SORTDOWN
Keep the linked list sorted by key in descending order.

Additionally, there are preprocessor flags for obtaining debugging printout (**−DDEBUG**) and for including a test driver in the calling routine (**−DDRIVER**). The source code should be consulted for further details.

EXAMPLE

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it out.

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

struct info {              /* this is the info stored in the table */
    int age, room;         /* other than the key. */
};
#define NUM_EMPL    5000    /* # of elements in search table */

main( )
{
        /* space to store strings */
        char string_space[NUM_EMPL * 20];
        /* space to store employee info */
        struct info info_space[NUM_EMPL];
        /* next avail space in string_space */
        char *str_ptr = string_space;
        /* next avail space in info_space */
        struct info *info_ptr = info_space;
        ENTRY item, *found_item, *hsearch( );
        /* name to look for in table */
        char name_to_find[30];
        int i = 0;

        /* create table */
        (void) hcreate(NUM_EMPL);
        while (scanf("%s%d%d", str_ptr, &info_ptr->age,
            &info_ptr->room) != EOF && i++ < NUM_EMPL) {
            /* put info in structure, and structure in item */
            item.key = str_ptr;
            item.data = (char *)info_ptr;
            str_ptr += strlen(str_ptr) + 1;
            info_ptr++;
            /* put item into table */
            (void) hsearch(item, ENTER);
        }

        /* access table */
        item.key = name_to_find;
```

```
        while (scanf("%s", item.key) != EOF) {
            if ((found_item = hsearch(item, FIND)) != NULL) {
            /* if item is in the table */
            (void)printf("found %s, age = %d, room = %d\n",
                found_item->key,
                ((struct info *)found_item->data)->age,
                ((struct info *)found_item->data)->room);
        } else {
            (void)printf("no such employee %s\n",
            name_to_find)
    }
  }
}
```

SEE ALSO

bsearch(3C), lsearch(3C), malloc(3C), string(3C), tsearch(3C).

DIAGNOSTICS

hsearch returns a NULL pointer if either the action is **FIND** and the item could not be found or the action is **ENTER** and the table is full.

hcreate returns zero if it cannot allocate sufficient space for the table.

WARNING

hsearch and *hcreate* use *malloc*(3C) to allocate space.

CAVEAT

Only one hash search table may be active at any given time.

NAME

hypot – Euclidean distance function

SYNOPSIS

```
#include <math.h>

double hypot (x, y)
  double x, y;
```

DESCRIPTION

hypot returns

sqrt(x * x + y * y),

taking precautions against unwarranted overflows.

DIAGNOSTICS

When the correct value would overflow, *hypot* returns **HUGE_VAL** and sets *errno* to **ERANGE**.

If x or y is NaN, NaN is returned and *errno* is set to EDOM.

This page is intentionally left blank

NAME

ioctl – control device

SYNOPSIS

int ioctl (fildes, request, arg)
int fildes, request;

DESCRIPTION

ioctl performs a variety of control functions on devices and STREAMS. For non-STREAMS files, the functions performed by this call are *device-specific* control functions. The arguments *request* and *arg* are passed to the file designated by *fildes* and are interpreted by the device driver. This control is infrequently used on non-STREAMS devices, with the basic input/output functions performed through the *read*(2) and *write*(2) system calls.

For STREAMS files, specific functions are performed by the *ioctl* call as described in *streamio*(7).

fildes is an open file descriptor that refers to a device. *request* selects the control function to be performed and will depend on the device being addressed. *arg* represents additional information that is needed by this specific device to perform the requested function. The data type of *arg* depends upon the particular control request, but it is either an integer or a pointer to a device-specific data structure.

In addition to device-specific and STREAMS functions, generic functions are provided by more than one device driver, for example, the general terminal interface [see *termio*(7)].

ioctl will fail for any type of file if one or more of the following are true:

[EACCES] Future error.

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

[EINTR] A signal was caught during the *ioctl* system call.

[ENOTTY] *fildes* is not associated with a device driver that accepts control functions.

ioctl will also fail if the device driver detects an error. In this case, the error is passed through *ioctl* without change to the caller. A particular driver might not have all of the following error cases. Other requests to device drivers will fail if one or more of the following are true:

[EFAULT] *request* requires a data transfer to or from a buffer pointed to by *arg*, but some part of the buffer is outside the process's allocated space.

[EINVAL] *request* or *arg* is not valid for this device.

[EIO] Some physical I/O error has occurred.

[ENXIO] The *request* and *arg* are valid for this device driver, but the service requested can not be performed on this particular subdevice.

[ENOLINK] *fildes* is on a remote machine and the link to that machine is no longer active.

STREAMS errors are described in *streamio*(7).

SEE ALSO

streamio(7), termio(7).

DIAGNOSTICS

Upon successful completion, the value returned depends upon the device control function, but must be a non-negative integer. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

isnan – test for NaN

SYNOPSIS

#include <math.h>

int isnan(x)
double x;

DESCRIPTION

The *isnan*( ) function tests whether *x* is NaN.

RETURN VALUE

The *isnan*( ) function returns non-zero if *x* is NaN. Otherwise, zero is returned.

ERRORS

No errors are defined.

This page is intentionally left blank

NAME

kill – send a signal to a process or a group of processes

SYNOPSIS

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

int kill (pid, sig)
pid_t pid;
int sig

DESCRIPTION

kill sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in *signal*(2), or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the effective user ID of the sending process is super-user.

If *pid* is greater than zero, *sig* will be sent to the process whose process ID is equal to *pid*. *pid* may equal 1, but signals may not be sent to other special processes [see *intro*(2)].

If *pid* is 0, *sig* will be sent to all processes (excluding special processes) whose process group ID is equal to the process group ID of the sender.

If *pid* is −1 and the effective user ID of the sender is not super-user, *sig* will be sent to all processes (excluding special processes) whose real user ID is equal to the effective user ID of the sender.

If *pid* is −1 and the effective user ID of the sender is super-user, *sig* will be sent to all processes (excluding special processes).

If *pid* is negative but not -1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

kill will fail and no signal will be sent if one or more of the following are true:

[EINVAL]	*sig* is not a valid signal number.
[EPERM]	*pid* specifies a special process except process 1, or *sig* is SIGKILL and *pid* is 1,
[EPERM]	The user ID of the sending process is not super-user, and its real or effective user ID does not match the real or effective user ID of the receiving process.
[ESRCH]	No process can be found corresponding to that specified by *pid*.

SEE ALSO

kill(1), getpid(2), setpgrp(2), signal(2), sigset(2).

DIAGNOSTICS

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

NAME

l3tol, ltol3 – convert between 3-byte integers and long integers

SYNOPSIS

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

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

DESCRIPTION

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

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

These functions are useful for file-system maintenance where the block numbers are three bytes long.

SEE ALSO

fs(4).

CAVEAT

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

This page is intentionally left blank

NAME

ldahread – read the archive header of a member of an archive file

SYNOPSIS

```
#include <ldfcn.h>

int ldahread (ldptr, arhead)
 LDFILE * ldptr;
 ARCHDR * arhead;
```

DESCRIPTION

If **TYPE(***ldptr***)** is the archive file magic number, *ldahread* reads the archive header of the common object file currently associated with *ldptr* into the area of memory beginning at *arhead*.

ldahread returns **SUCCESS** or **FAILURE**. *ldahread* will fail if **TYPE(***ldptr***)** does not represent an archive file, or if it cannot read the archive header.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ar(4), ldfcn(4).

This page is intentionally left blank

NAME

ldclose, ldaclose – close a common object file

SYNOPSIS

#include <ldfcn.h>

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;

DESCRIPTION

ldopen(3X) and *ldclose* are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of common object files can be processed as if it were a series of simple common object files.

If **TYPE(***ldptr***)** does not represent an archive file, *ldclose* will close the file and free the memory allocated to the **LDFILE** structure associated with *ldptr*. If **TYPE(***ldptr***)** is the magic number of an archive file, and if there are any more files in the archive, *ldclose* will reinitialize **OFFSET(***ldptr***)** to the file address of the next archive member and return **FAILURE**. The **LDFILE** structure is prepared for a subsequent *ldopen*(3X). In all other cases, *ldclose* returns **SUCCESS**.

ldaclose closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr* regardless of the value of **TYPE***(ldptr)*. *ldaclose* always returns **SUCCESS**. The function is often used in conjunction with *ldaopen*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fclose(3S), ldopen(3X), ldfcn(4).

This page is intentionally left blank

NAME

ldfhread – read the file header of a common object file

SYNOPSIS

#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE * ldptr;
FILHDR * filehead;

DESCRIPTION

ldfhread reads the file header of the common object file currently associated with *ldptr* into the area of memory beginning at *filehead*.

ldfhread returns **SUCCESS** or **FAILURE**. *ldfhread* will fail if it cannot read the file header.

In most cases the use of *ldfhread* can be avoided by using the macro **HEADER(***ldptr***)** defined in **ldfcn.h** [see ldfcn (4)]. The information in any field, *fieldname*, of the file header may be accessed using **HEADER(ldptr).fieldname**.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

This page is intentionally left blank

NAME

ldgetname – retrieve symbol name for common object file symbol table entry

SYNOPSIS

#include <ldfcn.h>

char *ldgetname (ldptr, symbol)
LDFILE *ldptr;
SYMENT *symbol;

DESCRIPTION

ldgetname returns a pointer to the name associated with **symbol** as a string. The string is contained in a static buffer local to *ldgetname* that is overwritten by each call to *ldgetname*, and therefore must be copied by the caller if the name is to be saved.

ldgetname can be used to retrieve names from object files without any backward compatibility problems. *ldgetname* will return NULL (defined in **stdio.h**) for an object file if the name cannot be retrieved. This situation can occur:

- if the "string table" cannot be found,
- if not enough memory can be allocated for the string table,
- if the string table appears not to be a string table (for example, if an auxiliary entry is handed to *ldgetname* that looks like a reference to a name in a nonexistent string table), or
- if the name's offset into the string table is past the end of the string table.

Typically, *ldgetname* will be called immediately after a successful call to *ldtbread* to retrieve the name associated with the symbol table entry filled by *ldtbread*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4).

NAME

ldlread, ldlinit, ldlitem – manipulate line number entries of a common object file function

SYNOPSIS

```
#include <ldfcn.h>

int ldlread(ldptr, fcnindx, linenum, linent)
 LDFILE *ldptr;
 long fcnindx;
 unsigned short linenum;
 LINENO *linent;

int ldlinit(ldptr, fcnindx)
 LDFILE *ldptr;
 long fcnindx;

int ldlitem(ldptr, linenum, linent)
 LDFILE *ldptr;
 unsigned short linenum;
 LINENO *linent;
```

DESCRIPTION

ldlread searches the line number entries of the common object file currently associated with *ldptr*. *ldlread* begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by *fcnindx*, the index of its entry in the object file symbol table. *ldlread* reads the entry with the smallest line number equal to or greater than *linenum* into the memory beginning at *linent*.

ldlinit and *ldlitem* together perform exactly the same function as *ldlread*. After an initial call to *ldlread* or *ldlinit, ldlitem* may be used to retrieve a series of line number entries associated with a single function. *ldlinit* simply locates the line number entries for the function identified by *fcnindx*. *ldlitem* finds and reads the entry with the smallest line number equal to or greater than *linenum* into the memory beginning at *linent*.

ldlread, *ldlinit*, and *ldlitem* each return either **SUCCESS** or **FAILURE**. *ldlread* will fail if there are no line number entries in the object file, if *fcnindx* does not index a function entry in the symbol table, or if it finds no line number equal to or greater than *linenum*. *ldlinit* will fail if there are no line number entries in the object file or if *fcnindx* does not index a function entry in the symbol table. *ldlitem* will fail if it finds no line number equal to or greater than *linenum*.

The programs must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbindex(3X), ldfcn(4).

NAME

ldlseek, ldnlseek – seek to line number entries of a section of a common object file

SYNOPSIS

#include <ldfcn.h>

int ldlseek (ldptr, sectindx)
LDFILE * ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE * ldptr;
char * sectname;

DESCRIPTION

ldlseek seeks to the line number entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

ldnlseek seeks to the line number entries of the section specified by *sectname*.

ldlseek and *ldnlseek* return **SUCCESS** or **FAILURE**. *ldlseek* will fail if *sectindx* is greater than the number of sections in the object file; *ldnlseek* will fail if there is no section name corresponding with * *sectname*. Either function will fail if the specified section has no line number entries or if it cannot seek to the specified line number entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

This page is intentionally left blank

NAME

ldohseek – seek to the optional file header of a common object file

SYNOPSIS

#include <ldfcn.h>

int ldohseek (ldptr)
LDFILE * ldptr;

DESCRIPTION

ldohseek seeks to the optional file header of the common object file currently associated with *ldptr*.

ldohseek returns **SUCCESS** or **FAILURE**. *ldohseek* will fail if the object file has no optional header or if it cannot seek to the optional header.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldfhread(3X), ldopen(3X), ldfcn(4).

This page is intentionally left blank

NAME

ldopen, ldaopen – open a common object file for reading

SYNOPSIS

```
#include <ldfcn.h>

LDFILE *ldopen (filename, ldptr)
 char *filename;
 LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
 char *filename;
 LDFILE *oldptr;
```

DESCRIPTION

ldopen and *ldclose*(3X) are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of common object files can be processed as if it were a series of simple common object files.

If *ldptr* has the value **NULL**, then *ldopen* will open *filename* and allocate and initialize the **LDFILE** structure, and return a pointer to the structure to the calling program.

If *ldptr* is valid and if **TYPE(***ldptr***)** is the archive magic number, *ldopen* will reinitialize the **LDFILE** structure for the next archive member of *filename*.

ldopen and *ldclose*(3X) are designed to work in concert. *ldclose* will return **FAILURE** only when **TYPE(***ldptr***)** is the archive magic number and there is another file in the archive to be processed. Only then should *ldopen* be called with the current value of *ldptr*. In all other cases, in particular whenever a new *filename* is opened, *ldopen* should be called with a **NULL** *ldptr* argument.

The following is a prototype for the use of *ldopen* and *ldclose*(3X).

```
/* for each filename to be processed */

ldptr = NULL;
do
{
    if ( (ldptr = ldopen(filename, ldptr)) != NULL )
    {
        /* check magic number */
        /* process the file */
    }
} while (ldclose(ldptr) == FAILURE );
```

If the value of *oldptr* is not **NULL**, *ldaopen* will open *filename* anew and allocate and initialize a new **LDFILE** structure, copying the **TYPE**, **OFFSET**, and **HEADER" "** fields from *oldptr*. *ldaopen* returns a pointer to the new **LDFILE** structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

Both *ldopen* and *ldaopen* open *filename* for reading. Both functions return **NULL** if *filename* cannot be opened, or if memory for the **LDFILE** structure cannot be allocated. A successful open does not insure that the given file is a common object file or an archived object file.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fopen(3S), ldclose(3X), ldfcn(4).

NAME

ldrseek, ldnrseek – seek to relocation entries of a section of a common object file

SYNOPSIS

#include <ldfcn.h>

int ldrseek (ldptr, sectindx)
LDFILE * ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE * ldptr;
char * sectname;

DESCRIPTION

ldrseek seeks to the relocation entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

ldnrseek seeks to the relocation entries of the section specified by *sectname*.

ldrseek and *ldnrseek* return **SUCCESS** or **FAILURE**. *ldrseek* will fail if *sectindx* is greater than the number of sections in the object file; *ldnrseek* will fail if there is no section name corresponding with *sectname*. Either function will fail if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

This page is intentionally left blank

NAME

ldshread, ldnshread – read an indexed/named section header of a common object file

SYNOPSIS

```
#include <ldfcn.h>

int ldshread (ldptr, sectindx, secthead)
 LDFILE *ldptr;
 unsigned short sectindx;
 SCNHDR *secthead;

int ldnshread (ldptr, sectname, secthead)
 LDFILE *ldptr;
 char *sectname;
 SCNHDR *secthead;
```

DESCRIPTION

ldshread reads the section header specified by *sectindx* of the common object file currently associated with *ldptr* into the area of memory beginning at *secthead*.

ldnshread reads the section header specified by *sectname* into the area of memory beginning at *secthead*.

ldshread and *ldnshread* return **SUCCESS** or **FAILURE**. *ldshread* will fail if *sectindx* is greater than the number of sections in the object file; *ldnshread* will fail if there is no section name corresponding with *sectname*. Either function will fail if it cannot read the specified section header.

Note that the first section header has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

This page is intentionally left blank

NAME

ldsseek, ldnsseek – seek to an indexed/named section of a common object file

SYNOPSIS

```
#include <ldfcn.h>

int ldsseek (ldptr, sectindx)
 LDFILE *ldptr;
 unsigned short sectindx;

int ldnsseek (ldptr, sectname)
 LDFILE *ldptr;
 char *sectname;
```

DESCRIPTION

ldsseek seeks to the section specified by *sectindx* of the common object file currently associated with *ldptr*.

ldnsseek seeks to the section specified by *sectname*.

ldsseek and *ldnsseek* return **SUCCESS** or **FAILURE**. *ldsseek* will fail if *sectindx* is greater than the number of sections in the object file; *ldnsseek* will fail if there is no section name corresponding with *sectname*. Either function will fail if there is no section data for the specified section or if it cannot seek to the specified section.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

This page is intentionally left blank

NAME

ldtbindex – compute the index of a symbol table entry of a common object file

SYNOPSIS

#include <ldfcn.h>

long ldtbindex (ldptr)
LDFILE * ldptr;

DESCRIPTION

ldtbindex returns the (**long**) index of the symbol table entry at the current position of the common object file associated with *ldptr*.

The index returned by *ldtbindex* may be used in subsequent calls to *ldtbread*(3X). However, since *ldtbindex* returns the index of the symbol table entry that begins at the current position of the object file, if *ldtbindex* is called immediately after a particular symbol table entry has been read, it will return the index of the next entry.

ldtbindex will fail if there are no symbols in the object file, or if the object file is not positioned at the beginning of a symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4).

This page is intentionally left blank

NAME

ldtbread – read an indexed symbol table entry of a common object file

SYNOPSIS

#include <ldfcn.h>

```
int ldtbread (ldptr, symindex, symbol)
 LDFILE *ldptr;
 long symindex;
 SYMENT *symbol;
```

DESCRIPTION

ldtbread reads the symbol table entry specified by *symindex* of the common object file currently associated with *ldptr* into the area of memory beginning at **symbol**.

ldtbread returns **SUCCESS** or **FAILURE**. *ldtbread* will fail if *symindex* is greater than or equal to the number of symbols in the object file, or if it cannot read the specified symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldgetname(3x), ldopen(3X), ldtbseek(3X), ldfcn(4).

This page is intentionally left blank

NAME

ldtbseek – seek to the symbol table of a common object file

SYNOPSIS

#include <ldfcn.h>

int ldtbseek (ldptr)
LDFILE * ldptr;

DESCRIPTION

ldtbseek seeks to the symbol table of the common object file currently associated with *ldptr*.

ldtbseek returns **SUCCESS** or **FAILURE**. *ldtbseek* will fail if the symbol table has been stripped from the object file, or if it cannot seek to the symbol table.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldfcn(4).

This page is intentionally left blank

NAME

link – link to a file

SYNOPSIS

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

DESCRIPTION

path1 points to a path name naming an existing file. *path2* points to a path name naming the new directory entry to be created. *link* creates a new link (directory entry) for the existing file.

Upon successful completion, the *link*() function will mark for update the *st_ctime* field of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new entry are marked for update.

link will fail and no link will be created if one or more of the following are true:

[EACCES]	A component of either path prefix denies search permission.
[EACCES]	The requested link requires writing in a directory with a mode that denies write permission.
[EEXIST]	The link named by *path2* exists.
[EFAULT]	*path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *link* system call.
[EMLINK]	The maximum number of links LINK_MAX to a file would be exceeded.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

[ENAMETOOLONG]	The length of the *path1* or path2 string exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX} and {_POSIX_NO_TRUNC} is in effect.
[ENOENT]	A component of either path prefix does not exist.
[ENOENT]	The file named by *path1* does not exist.
[ENOENT]	*path2* points to a null path name.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[ENOTDIR]	A component of either path prefix is not a directory.
[EPERM]	The file named by *path1* is a directory and the effective user ID is not super-user.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EXDEV]	The link named by *path2* and the file named by *path1* are on different logical devices (file systems).

SEE ALSO

unlink(2).

DIAGNOSTICS

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

NAME

localeconv – get numeric formatting information

SYNOPSIS

#include <locale.h>

struct lconv * localeconv (void);

DESCRIPTION

localeconv sets the components of an object with type *struct lconv* (defined in *locale.h*) with the values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale [see *setlocale*(3C)].

The definition of *struct lconv* is given below (the values for the fields in the "C" locale are given in comments):

```
char *decimal_point;      /* "." */
char *thousands_sep;      /* "" (zero length string) */
char *grouping;           /* "" */
char *int_curr_symbol;    /* "" */
char *currency_symbol;    /* "" */
char *mon_decimal_point;  /* "" */
char *mon_thousands_sep;  /* "" */
char *mon_grouping;       /* "" */
char *positive_sign;      /* "" */
char *negative_sign;      /* "" */
char int_frac_digits;     /* CHAR_MAX */
char frac_digits;         /* CHAR_MAX */
char p_cs_precedes;       /* CHAR_MAX */
char p_sep_by_space;      /* CHAR_MAX */
char n_cs_precedes;       /* CHAR_MAX */
char n_sep_by_space;      /* CHAR_MAX */
char p_sign_posn;         /* CHAR_MAX */
char n_sign_posn;         /* CHAR_MAX */
```

The members of the structure with type char * are strings, any of which (except decimal_point) can point to "", to indicate that the value is not available in the current locale or is of zero length. The members with type char are nonnegative numbers, any of which can be **CHAR_MAX** (defined in the *limits.h* header file) to indicate that the value is not available in the current locale. The members are the following:

`char *decimal_point`

The decimal-point character used to format non-monetary quantities.

`char *thousands_sep`

The character used to separate groups of digits to the left of the decimal-point character in formatted non-monetary quantities.

`char *grouping`

A string in which each element is taken as an integer that indicates the number of digits that comprise the current group in a formatted non-monetary quantity. The elements of grouping are interpreted according to the following:

CHAR–MAX	No further grouping is to be performed.
0	The previous element is to be repeatedly used for the remainder of the digits.
other	The value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.

`char *int_curr_symbol`

The international currency symbol applicable to the current locale, left-justified within a four-character space-padded field. The character sequences should match with those specified in: *ISO 4217 Codes for the Representation of Currency and Funds*.

`char *currency_symbol`

The local currency symbol applicable to the current locale.

`char *mon_decimal_point`

The decimal-point used to format monetary quantities.

`char *mon_thousands_sep`

The separator for groups of digits to the left of the decimal-point in formatted monetary quantities.

`char *mon_grouping`

A string in which each element is taken as an integer that indicates the number of digits that comprise the current group in a formatted monetary quantity. The elements of mon_grouping are interpreted according to the rules described under grouping.

`char *positive_sign`

The string used to indicate a nonnegative-valued formatted monetary quantity.

`char *negative_sign`

The string used to indicate a negative-valued formatted monetary quantity.

`char int_frac_digits`

The number of fractional digits (those to the right of the decimal-point) to be displayed in an internationally formatted monetary quantity.

`char frac_digits`

The number of fractional digits (those to the right of the decimal-point) to be displayed in a formatted monetary quantity.

`char p_cs_precedes`

Set to 1 or 0 if the currency_symbol respectively precedes or succeeds the value for a nonnegative formatted monetary quantity.

`char p_sep_by_space`

Set to 1 or 0 if the currency_symbol respectively is or is not separated by a space from the value for a nonnegative formatted monetary quantity.

char n_cs_precedes

Set to 1 or 0 if the currency_symbol respectively precedes or succeeds the value for a negative formatted monetary quantity.

char n_sep_by_space

Set to 1 or 0 if the currency_symbol respectively is or is not separated by a space from the value for a negative formatted monetary quantity.

char p_sign_posn

Set to a value indicating the positioning of the positive_sign for a nonnegative formatted monetary quantity. The value of p_sign_posn is interpreted according to the following:

0 Parentheses surround the quantity and currency_symbol.

1 The sign string precedes the quantity and currency_symbol.

2 The sign string succeeds the quantity and currency_symbol.

3 The sign string immediately precedes the currency_symbol.

4 The sign string immediately succeeds the currency_symbol.

char n_sign_posn

Set to a value indicating the positioning of the negative_sign for a negative formatted monetary quantity. The value of n_sign_posn is interpreted according to the rules described under p_sign_posn.

RETURNS

localeconv returns a pointer to the filled-in object. The structure pointed to by the return value may be overwritten by a subsequent call to localeconv.

EXAMPLES

The following table illustrates the rules used by four countries to format monetary quantities.

Country	Positive format	Negative format	International format
Italy	L.1.234	-L.1.234	ITL.1.234
Netherlands	F 1.234,56	F -1.234,56	NLG 1.234,56
Norway	kr1.234,56	kr1.234,56-	NOK 1.234,56
Switzerland	SFrs.1,234.56	SFrs.1,234.56C	CHF 1,234.56

For these four countries, the respective values for the monetary members of the structure returned by *localeconv* are as follows:

	Italy	Netherlands	Norway	Switzerland
int_curr_symbol	"ITL."	"NLG "	"NOK "	"CHF "
currency_symbol	"L."	"F"	"kr"	"SFrs."
mon_decimal_point	""	","	","	"."
mon_thousands_sep	"."	"."	"."	","
mon_grouping	"\3"	"\3"	"\3"	"\3"
positive_sign	""	""	""	""
negative_sign	"-"	"-"	"-"	"C"
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_space	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2

FILES

/usr/lib/locale/*locale*/LC_MONETARY
LC_MONETARY database for *locale*.

/usr/lib/locale/*locale*/LC_NUMERIC
LC_NUMERIC database for *locale*.

SEE ALSO

chrtbl(1M), montbl(1M), setlocale(3C) in the *System V Reference Manual.*

NAME

lockf – record locking on files

SYNOPSIS

```
#include <unistd.h>

int lockf (fildes, function, size)
  long size;
  int fildes, function;
```

DESCRIPTION

The *lockf* command will allow sections of a file to be locked; advisory or mandatory write locks depending on the mode bits of the file [see *chmod*(2)]. Locking calls from other processes which attempt to lock the locked file section will either return an error value or be put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. [See *fcntl*(2) for more information about record locking.]

fildes is an open file descriptor. The file descriptor must have O_WRONLY or O_RDWR permission in order to establish lock with this function call.

function is a control value which specifies the action to be taken. The permissible values for *function* are defined in **<unistd.h>** as follows:

```
#define F_ULOCK 0/* Unlock a previously locked section */
#define F_LOCK  1/* Lock a section for exclusive use */
#define F_TLOCK 2/* Test and lock a section for exclusive use */
#define F_TEST  3/* Test section for other processes locks */
```

All other values of *function* are reserved for future extensions and will result in an error return if not implemented.

F_TEST is used to detect if a lock by another process is present on the specified section. F_LOCK and F_TLOCK both lock a section of a file if the section is available. F_ULOCK removes locks from a section of the file.

size is the number of contiguous bytes to be locked or unlocked. The resource to be locked starts at the current offset in the file and extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is zero, the section from the current offset through the largest file offset is locked (i.e., from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked as such locks may exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent sections occur, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

F_LOCK and F_TLOCK requests differ only by the action taken if the resource is not available. F_LOCK will cause the calling process to sleep until the resource is available. F_TLOCK will cause the function to return a −1 and set *errno* to [EACCES] error if the section is already locked by another process.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an [EDEADLK] error is returned and the requested section is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing another process's locked resource. Thus calls to *lockf* or *fcntl* scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The *alarm*(2) command may be used to provide a timeout facility in applications which require this facility.

The *lockf* utility will fail if one or more of the following are true:

[EACCES] *cmd* is F_TLOCK or F_TEST and the section is already locked by another process.

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

[ECOMM] *fildes* is on a remote machine and the link to that machine is no longer active.

[EDEADLK] *cmd* is F_LOCK and a deadlock would occur. Also the *cmd* is either F_LOCK, F_TLOCK, or F_ULOCK and the number of entries in the lock table would exceed the number allocated on the system.

SEE ALSO

chmod(2), close(2), creat(2), fcntl(2), intro(2), open(2), read(2), write(2).

DIAGNOSTICS

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

WARNINGS

Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The standard I/O package is the most common source of unexpected buffering.

Because in the future the variable *errno* will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value.

This page is intentionally left blank

NAME

logname – return login name of user

SYNOPSIS

char *logname( )

DESCRIPTION

logname returns a pointer to the null-terminated login name; it extracts the **LOGNAME** environment variable from the user's environment.

This routine is kept in **/lib/libPW.a**.

The program must be loaded with the object file access routine library **libPW.a.**.

FILES

/etc/profile

SEE ALSO

env(1), login(1), getenv(3C), profile(4), environ(5).

CAVEATS

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

This method of determining a login name is subject to forgery.

This page is intentionally left blank

NAME

lsearch, lfind – linear search and update

SYNOPSIS

```
#include <search.h>

void *lsearch (key, base, nelp, width, compar)
void *base, *key;
size_t width, *nelp;
int (*compar)( );

void *lfind (key, base, nelp, width, compar)
void *base, *key;
size_t width, *nelp;
int (*compar)( );
```

DESCRIPTION

lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. *key* points to the datum to be sought in the table. *base* points to the first element in the table.

nelp points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table.

compar is the name of the comparison function which the user must supply (*strcmp*, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

lfind is the same as *lsearch* except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-void. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared. Although declared as type pointer-to-void, the value returned should be cast into type pointer-to-element.

EXAMPLE

This fragment will read in less than TABSIZE strings of length less than ELSIZE and store them in a table, eliminating duplicates.

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

#define TABSIZE 50
#define ELSIZE 120

. . .
    char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
    size_t nel = 0;
    . . .
    while (fgets(line, ELSIZE, stdin) != NULL &&
      nel < TABSIZE)
        (void) lsearch((void *)line, (void *)tab, &nel,
              ELSIZE, strcmp);
    . . .
```

SEE ALSO

bsearch(3C), hsearch(3C), string(3C), tsearch(3C).

DIAGNOSTICS

If the searched for datum is found, both *lsearch* and *lfind* return a pointer to it. Otherwise, *lfind* returns NULL and *lsearch* returns a pointer to the newly added element.

BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

NAME

lseek – move read/write file pointer

SYNOPSIS

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

off_t lseek (fildes, offset, whence)
  int fildes;
  off_t offset;
  int whence;

DESCRIPTION

fildes is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call. *lseek* sets the file pointer associated with *fildes* as follows:

If *whence* is SEEK_SET (0), the pointer is set to *offset* bytes.

If *whence* is SEEK_CUR (1), the pointer is set to its current location plus *offset*.

If *whence* is SEEK_END (2), the pointer is set to the size of the file plus *offset*.

The symbolic values of *whence* are found in the **<unistd.h>** header file.

Upon successful completion, the resulting pointer location, as measured in bytes from the beginning of the file, is returned.

lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF]	*fildes* is not an open file descriptor.
[EINVAL]	*whence* is not 0, 1, or 2.
[EINVAL]	The resulting file pointer would be negative.
[ESPIPE]	*fildes* is associated with a pipe or fifo.

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

SEE ALSO

creat(2), dup(2), fcntl(2), open(2).

DIAGNOSTICS

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

NAME

admin – create and administer SCCS files

SYNOPSIS

admin [–**n**] [–**i**[name]] [–**r**rel] [–**t**[name]] [–**f**flag[flag-val]] [–**d**flag[flag-val]] [–**a**login] [–**e**login] [–**m**[mrlist]] [–**y**[comment]] [–**h**] [–**z**] files

DESCRIPTION

admin is used to create new SCCS files and change parameters of existing ones. Arguments to *admin*, which may appear in any order, consist of keyletter arguments, which begin with –, and named files (note that SCCS file names must begin with the characters **s.**). If a named file does not exist, it is created, and its parameters are initialized according to the specified keyletter arguments. Parameters not initialized by a keyletter argument are assigned a default value. If a named file does exist, parameters corresponding to specified keyletter arguments are changed, and other parameters are left as is.

If a directory is named, *admin* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed since the effects of the arguments apply independently to each named file.

–**n** This keyletter indicates that a new SCCS file is to be created.

–**i***[name]* The *name* of a file from which the text for a new SCCS file is to be taken. The text constitutes the first delta of the file (see –**r** keyletter for delta numbering scheme). If the **i** keyletter is used, but the file name is omitted, the text is obtained by reading the standard input until an end-of-file is encountered. If this keyletter is omitted, then the SCCS file is created empty.

Only one SCCS file may be created by an *admin* command on which the **i** keyletter is supplied. Using a single *admin* to create two or more SCCS files requires that they be created empty (no **−i** keyletter). Note that the **−i** keyletter implies the **−n** keyletter.

−r*rel* The *rel*ease into which the initial delta is inserted. This keyletter may be used only if the **−i** keyletter is also used. If the **−r** keyletter is not used, the initial delta is inserted into release 1. The level of the initial delta is always 1 (by default initial deltas are named 1.1).

−t*[name]* The *name* of a file from which descriptive text for the SCCS file is to be taken. If the **−t** keyletter is used and *admin* is creating a new SCCS file (the **−n** and/or **−i** keyletters also used), the descriptive text file name must also be supplied.

In the case of existing SCCS files: (1) a **−t** keyletter without a file name causes removal of descriptive text (if any) currently in the SCCS file, and (2) a **−t** keyletter with a file name causes text (if any) in the named file to replace the descriptive text (if any) currently in the SCCS file.

−f*flag* This keyletter specifies a *flag*, and, possibly, a value for the *flag*, to be placed in the SCCS file.

Several **f** keyletters may be supplied on a single *admin* command line. The allowable *flag*s and their values are:

b Allows use of the **−b** keyletter on a *get*(1) command to create branch deltas.

c*ceil* The highest release (i.e., "ceiling"), a number greater than 0 but less than or equal to 9999, which may be retrieved by a *get*(1) command for editing. The default value for an unspecified **c** flag is 9999.

f*floor* The lowest release (i.e., "floor"), a number greater than 0 but less than 9999, which may be retrieved by a *get*(1) command for editing. The default value for an unspecified **f** flag is 1.

d*SID* The default delta number (SID) to be used by a *get*(1) command.

i*[str]* Causes the "No id keywords (ge6)" message issued by *get*(1) or *delta*(1) to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords [see *get*(1)] are found in the text retrieved or stored in the SCCS file. If a value is supplied, the keywords must exactly match the given string, however the string must contain a keyword, and no embedded newlines.

j Allows concurrent *get*(1) commands for editing on the same SID of an SCCS file. This allows multiple concurrent updates to the same version of the SCCS file.

l*list* A *list* of releases to which deltas can no longer be made (**get −e** against one of these "locked" releases fails). The *list* has the following syntax:

<list> ::= <range> | <list> , <range>

<range> ::= (R) | **a**

The character **a** in the *list* is equivalent to specifying *all releases* for the named SCCS file.

n Causes *delta*(1) to create a "null" delta in each of those releases (if any) being skipped when a delta is made in a *new* release (e.g., in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as "anchor points" so that branch deltas may later be created from them. The absence of this flag causes skipped releases to be non-existent in the SCCS file, preventing branch deltas from being created from them in the future.

q*text* User definable text substituted for all occurrences of the %Q% keyword in SCCS file text retrieved by *get*(1).

m*mod* *Mod*ule name of the SCCS file substituted for all occurrences of the %M% keyword in SCCS file text retrieved by *get*(1). If the **m** flag is not specified, the value assigned is the name of the SCCS file with the leading **s.** removed.

t*type* *Type* of module in the SCCS file substituted for all occurrences of %Y% keyword in SCCS file text retrieved by *get*(1).

v*pgm* Causes *delta*(1) to prompt for Modification Request (MR) numbers as the reason for creating a delta. The optional value specifies the name of an MR number validity checking program [see *delta*(1)]. (If this flag is set when creating an SCCS file, the **m** keyletter must also be used even if its value is null).

−d*flag* Causes removal (deletion) of the specified *flag* from an SCCS file. The **−d** keyletter may be specified only when processing existing SCCS files. Several **−d** keyletters may be supplied on a single *admin* command. See the **−f** keyletter for allowable *flag* names.

l*list* — A *list* of releases to be "unlocked". See the **−f** keyletter for a description of the **l** flag and the syntax of a *list*.

−a*login* — A *login* name, or numerical UNIX system group ID, to be added to the list of users which may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all *login* names common to that group ID. Several **a** keyletters may be used on a single *admin* command line. As many *login*s, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas. If *login* or group ID is preceded by a **!** they are to be denied permission to make deltas.

−e*login* — A *login* name, or numerical group ID, to be erased from the list of users allowed to make deltas (changes) to the SCCS file. Specifying a group ID is equivalent to specifying all *login* names common to that group ID. Several **e** keyletters may be used on a single *admin* command line.

−m*[mrlist]* — The list of Modification Requests (MR) numbers is inserted into the SCCS file as the reason for creating the initial delta in a manner identical to *delta*(1). The **v** flag must be set and the MR numbers are validated if the **v** flag has a value (the name of an MR number validation program). Diagnostics will occur if the **v** flag is not set or MR validation fails.

−y*[comment]* The *comment* text is inserted into the SCCS file as a comment for the initial delta in a manner identical to that of *delta*(1). Omission of the **−y** keyletter results in a default comment line being inserted in the form:

date and time created *YY/MM/DD HH:MM:SS* by *login*

The **−y** keyletter is valid only if the **−i** and/or **−n** keyletters are specified (i.e., a new SCCS file is being created).

−h Causes *admin* to check the structure of the SCCS file [see *sccsfile*(5)], and to compare a newly computed check-sum (the sum of all the characters in the SCCS file except those in the first line) with the check-sum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced. This keyletter inhibits writing on the file, so that it nullifies the effect of any other keyletters supplied, and is, therefore, only meaningful when processing existing files.

−z The SCCS file check-sum is recomputed and stored in the first line of the SCCS file (see **−h**, above).

Note that use of this keyletter on a truly corrupted file may prevent future detection of the corruption.

The last component of all SCCS file names must be of the form **s.***file-name*. New SCCS files are given mode 444 [see *chmod*(1)]. Write permission in the pertinent directory is, of course, required to create a file. All writing done by *admin* is to a temporary x-file, called **x.***file-name*, [see *get*(1)], created with mode 444 if the *admin* command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of *admin*, the SCCS file is removed (if it exists), and the x-file is renamed with the name of the SCCS file. This ensures that changes are made to the SCCS file only if no errors occurred.

It is recommended that directories containing SCCS files be mode 755 and that SCCS files themselves be mode 444. The mode of the directories allows only the owner to modify SCCS files contained in the directories. The mode of the SCCS files prevents any modification at all except by SCCS commands.

If it should be necessary to patch an SCCS file for any reason, the mode may be changed to 644 by the owner allowing use of *ed*(1). *Care must be taken!* The edited file should *always* be processed by an **admin −h** to check for corruption followed by an **admin −z** to generate a proper check-sum. Another **admin −h** is recommended to ensure the SCCS file is valid.

admin also makes use of a transient lock file (called **z.***file-name*), which is used to prevent simultaneous updates to the SCCS file by different users. See *get*(1) for further information.

FILES

g − file	Existed before the execution of *delta*; removed after completion of *delta*.
p − file	Existed before the execution of *delta*; may exist after completion of *delta*.
q − file	Created during the execution of *delta*; removed after completion of *delta*.
x − file	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z − file	Created during the execution of *delta*; removed during the execution of *delta*.

d−file	Created during the execution of *delta*; removed after completion of *delta*.
/usr/bin/bdiff	Program to compute differences between the "gotten" file and the *g-file*.

SEE ALSO

delta(1), ed(1), get(1), help(1), prs(1), what(1), sccsfile(4).

DIAGNOSTICS

Use *help*(1) for explanations.

This page is intentionally left blank

NAME

ar – archive and library maintainer for portable archives

SYNOPSIS

ar key [posname] afile [name] ...

DESCRIPTION

The *ar* command maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the link editor. It can be used, though, for any similar purpose. The magic string and the file headers used by *ar* consist of printable ASCII characters. If an archive is composed of printable files, the entire archive is printable.

When *ar* creates an archive, it creates headers in a format that is portable across all machines. The portable archive format and structure is described in detail in *ar*(4). The archive symbol table [described in *ar*(4)] is used by the link editor [*ld*(1)] to effect multiple passes over libraries of object files in an efficient manner. An archive symbol table is only created and maintained by *ar* when there is at least one object file in the archive. The archive symbol table is in a specially named file which is always the first file in the archive. This file is never mentioned or accessible to the user.

Whenever the *ar*(1) command is used to create or update the contents of such an archive, the symbol table is rebuilt. The **s** option described below will force the symbol table to be rebuilt.

Unlike command options, the command key is a required part of *ar*'s command line. The key (which may begin with a –) is formed with one of the following letters: **drqtpmx**. Arguments to the *key*, alternatively, are made with one of more of the following set: **vuaibcls**.

posname is an archive member name used as a reference point in positioning other files in the archive. *afile* is the archive file. The *name* are constituent files in the archive file.

The *ar* command determines the type of the inputfile, and will only accept files of the same type. The possible filetypes are text files, code files, and archives consisting of each file type types (see *intro*(1)).

The meanings of the *key* characters are as follows:

d Delete the named files from the archive file.

r Replace the named files in the archive file. If the optional character **u** is used with **r**, then only those files with dates of modification later than the archive files are replaced. If an optional positioning character from the set **abi** is used, then the *posname* argument must be present and specifies that new files are to be placed after (**a**) or before (**b** or **i**) *posname*. Otherwise new files are placed at the end.

q Quickly append the named files to the end of the archive file. Optional positioning characters are invalid. The command does not check whether the added members are already in the archive. This option is useful to avoid quadratic behavior when creating a large archive piece-by-piece. Unchecked, the file may grow exponentially up to the second degree.

t Print a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.

p Print the named files in the archive.

m Move the named files to the end of the archive. If a positioning character is present, then the *posname* argument must be present and, as in **r**, specifies where the files are to be moved.

x Extract the named files. If no names are given, all files in the archive are extracted. In neither case does **x** alter the archive file.

The meanings of the key arguments are as follows:

a Specifies that the file goes after the existing file (*afile*). Use this suboption with the **m** or **r** options.

b Specifies that the file goes before the existing file (*afile*). Use this suboption with the **m** or **r** options.

c Suppress the message that is produced by default when *afile* is created.

i Specifies that the file goes before the existing file (*afile*). Use this suboption with the **m** or **r** options.

l Place temporary files in the local (current working) directory rather than in the default temporary directory *TMPDIR*.

s Makes a symbol definition (symdef file) as the first file of an archive. If you specify ‘s’, the archiver creates the symdef file as its last action before finishing execution. You must specify at least one other archive option (**m, p, q, r,** or **t**) when you use the **s** option. Supermax RISC **ar** builds the symbol table by default.

o R3KMI- and R3KMO-type: Forces a newly created file to have the 'last-modified' date that it had before it was extracted from the archive. Use this option with the **x** option.

u Prevents the archiver from replacing an existing file unless the replacement is newer than the existing file. This option uses the UNIX system ‘last modified’ data for this comparison. Use this suboption with the **r** option.

v Give a verbose file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with **t**, give a long listing of all information about the files.

z Suppress symbol table building.

FILES

/bin/ar	The *ar* startup program.
/tmp/v*	temporary files.

SEE ALSO

ld(1), lorder(1), odump(1), strip(1), ranhash(3X), a.out(4), ar(4).

NOTES

If the same file is mentioned twice in an argument list, it may be put in the archive twice.

The **o** option does not change the 'last-modified' date of a file unless you own the extracted file or you are the super-user.

The **s** option is not operative as *ar* will always build the hash table by default unless the **z** option is used.

NAME

as – Assembler

SYNOPSIS

as [options ...] file

DESCRIPTION

as assembles the named file. It is recommended that the input source file ends with **.s**.

Before the input file is assembled it is processed by the C preprocessor – cpp.

The macros LANGUAGE_ASSEMBLY, mips, host_mips and unix are defined.

If the environment variable **TMPDIR** is set, the value is used as the directory to place any temporary files rather than the default **/tmp/**.

The following options are recognized by *as* **and have the same meaning in** *cc*(1):

–V This option gives current version number for the assembler.

–g0 Have the assembler produce no symbol table information for symbolic debugging. This is the default.

–g1 Have the assembler produce additional symbol table information for accurate but limited symbolic debugging of partially optimized code.

–g or **–g2**
Have the assembler produce additional symbol table information for full symbolic debugging and not do optimizations that limit full symbolic debugging.

–g3 Have the assembler produce additional symbol table information for full symbolic debugging for fully optimized code. This option makes the debugger inaccurate.

−o The default output file is *a.out* but can be overridden by giving the '**−o**' option.

−w Suppress warning messages.

−P Run only the C macro preprocessor and put the result in a file with the suffix of the source file changed to '**.i**' or if the file has no suffix then a '**.i**' is added to the source file name. The '**.i**' file has no '#' lines in it.

−E Run only the C macro preprocessor on the file and send the result to the standard output.

−D*name*=*def*
−D*name*
Define the *name* to the C macro preprocessor, as if by '#define'. If no definition is given, the name is defined as "1".

−U*name*
Remove any initial definition of *name*.

−I*dir*
'#include' files whose names do not begin with '/' are always sought first in the directory of the *file* argument, then in directories specified in **−I** options, and finally in the standard directory (**/usr/include**).

−G *num*
Specify the maximum size, in bytes, of a data item that is to be accessed from the global pointer. *num* is assumed to be a decimal number. If *num* is zero, no data is accessed from the global pointer. The default value for *num* is 8 bytes.

−v Print the passes as they execute with their arguments and their input and output files.

−nocpp
Do not run the C macro preprocessor on assembly source files before compiling.

−m Apply the M4 preprocessor to the source file before assembling it.

The options described below primarily aid compiler development and are not generally used:

−H*c* Halt compiling after the pass specified by the character *c*, producing an intermediate file for the next pass. The *c* can be **[a]**. It selects the assembler pass in the same way as the **−t** option. If this option is used, the symbol table file produced and used by the passes, is the last component of the source file with the suffix changed to '.T', or a '.T' is added if the source file has no suffix. This file is not removed.

−K Build and use intermediate file names with the last component of the source file's name replacing its suffix with the conventional suffix for the type of file (for example '.G' file for binary assembly language). If the source file has no suffix the conventional suffix is added to the source file name. These intermediate files are never removed even when a pass encounters a fatal error.

−W *c*[*c*...],*arg1*[,*arg2*...]
Pass the argument[s] *argi* to the compiler pass[es] *c*[*c*..]. The *c's* are one of **[pab]**. The c's selects the compiler pass in the same way as the **−t** option.

The options **−t**[**hpab**], **−h***path,* and **−B***string* select a name to use for a particular pass. These arguments are processed from left to right so their order is significant. When the **−B** option is encountered, the selection of names takes place using the last **−h** and **−t** options. Therefore, the **−B** option is always required when using **−h** or **−t**. Sets of these options can be used to select any combination of names.

−t[**hpab**]
Select the names. The names selected are those designated by the characters following the **−t** option according to the following table:

Name	Character	
include	h	(see note below)
cpp	p	
as0	a	
as1	b	

If the character 'h' is in the **−t** argument then a directory is added to the list of directories to be used in searching for '#include' files. This directory name has the form /usr/include*string*. The standard directory is still searched.

−h*path*
Use *path* rather than the directory where the name is normally found.

−B*string*
Append *string* to all names specified by the **−t** option. If no **−t** option has been processed before the **−B**, the **−t** option is assumed to be "hpab". This list designates all names.

FILES

/bin/as

$COMP_HOST_ROOT/usr/lib/cmplrs/as0
Symbolic to binary assembly language translator.

$COMP_HOST_ROOT/usr/lib/cmplrs/as1
Binary assembly language assembler and reorganizer.

SEE ALSO

cc(1), ld(1).

NAME

cb – C program beautifier

SYNOPSIS

cb [**–s**] [**–j**] [**–l** leng] [file ...]

DESCRIPTION

The *cb* command reads C programs either from its arguments or from the standard input, and writes them on the standard output with spacing and indentation that displays the structure of the code. During default options, *cb* preserves all user newlines.

cb accepts the following options.

–s Canonicalizes the code to the style of Kernighan and Ritchie in *The C Programming Language*.

–j Causes split lines to be put back together.

–l *leng* Causes *cb* to split lines that are longer than *leng*.

SEE ALSO

cc(1).

The C Programming Language. Prentice-Hall, 1978.

BUGS

Punctuation that is hidden in preprocessor statements will cause indentation errors.

This page is intentionally left blank

NAME

cc – Supermax RISC C–compiler

SYNOPSIS

cc [options] ... file

DESCRIPTION

cc is the C–compiler. Files whose names ends with **.c** are taken to be C–source programs. They are compiled and an object or executable file is produced. If an object file is produced, the name of the object file is the same as on the source file except that the **.c** is replaced with the deleted. If an executable file is produced, the default file name is **a.out**, or if the **–o** option is specified, the name is taken from the argument to the **–o** option.

File parameters ending with **.s** are taken to be assembly source code and are passed on to the assembler. File parameters ending with **.o** are taken to be object files and are passed on to the link editor.

The TARGETMC-environment controls the code generation (see *intro*(1)).

cc defines some default C preprocessor macros according to the TARGETMC-evironment.

The following macros are defined:

unix, **mips**, **supermax**, **host_mips**, **MIPSEB**, **SYSTYPE_SYSV**, **LANGUAGE_C**.

Following options are interpreted by **cc**.

–c Produce an **.o** object file and suppress the link edit phase rather than producing an executable program.

–g0 Have the compiler produce no symbol table information for symbolic debugging. This is the default.

–g1 Have the compiler produce additional symbol table information for accurate but limited symbolic debugging of partially optimized code.

−g or **−g2**
Have the compiler produce additional symbol table information for full symbolic debugging and not do optimizations that limit full symbolic debugging.

−g3 Have the compiler produce additional symbol table information for full symbolic debugging for fully optimized code. This option makes the debugger inaccurate.

−o *output*
Name the final output file *output*. If this option is used, the file 'a.out' is undisturbed.

−v Verbose. Print the name of each subprocess as it is executing.

−E Run only the C preprocessor, **cpp**, on the named files and send the output to the standard output.

−P Same as E option but leave the output on a file suffixed **.i**

−S Generate assembly source code file rather than an object or an executable file. The compiled C−program assembly file is suffixed **.s**.

−V Print current version number.

The following options are passed by *cc* (with their associated arguments) to the preprocessor phase:

−C By default, the preprocessor strips C−language style comments. If the C−options is specified, all comments (except those found on preprocessor directive lines) are passed along.

−D*name*=*def*
−D*name*
−D *name*=*def*
−D *name*
Define the *name* to the C macro preprocessor, as if by '#define'. If no definition is given, the name is defined as "1".

−U*name*
−U *name*
Remove any initial definition of *name*.

−I*dir*
−I *dir* Change the algorithm for searching for *#include* files whose names do not begin with / to look in *dir* before looking in the directories on the standard list. Thus, *#include* files whose names are enclosed in double quotes are searched for first in the directory of the *file* argument, then in directories names in *−I* options, and last in directories on a standard list. For *#include* files whose names are enclosed in < >, the directory of the *file* argument is not searched.

−L*dir*
−L *dir* Change the algorithm for searching for the library *xxx* to look in *dir* before looking in the default library directories. This option is only effective if it precedes the **−l** option on the command line, (see *ld*(1)).

−w Suppress warning messages.

−O0 Turn off all optimizations.

−O1 Turn on all optimizations that can be done quickly. This is the default.

−O or **−O2**
Invoke the global *ucode* optimizer.

−O3 Do all optimizations, including global register allocation. This option must precede all source file arguments. With this option, a *ucode* object file is created for each C source file and left in a '.u' file. The newly created ucode object files, the ucode object files specified on the command line and the runtime startup routine and all the runtime libraries are ucode linked. Optimization is done on the resulting ucode linked file and then it is linked as normal producing an "a.out" file. No resulting '.o' file is left from the ucode linked result as in previous releases. In fact **−c** can no longer be specified with −03.

–Olimit *num*

Specify the maximum size, in basic blocks, of a routine that will be optimized by the global optimizer. If a routine has more than this number of basic blocks it will not be optimized and a message will be printed. An option specifying that the global optimizer is to be run (**–O**, **–O2**, or **–O3**) must also be specified. *num* is assumed to be a decimal number. The default value for *num* is 500 basic blocks.

–edit[*0-9*]

Invoke the editor of choice (as defined by the environment variable EDITOR), or *vi*(1) (if EDITOR is not defined) when syntax or semantic errors are detected by the compiler's frontend. When compiling on a character based terminal, the compile job has to be in the foreground for this option to take effect. For compile jobs done on a window based terminal/workstation, this option would always take effect whether it is in the foreground or background. The editor is invoked with two files: the error message file and the source file. First use the error message file to locate the line numbers of all the errors, the switch to the source file to make corrections. Once you exit out of the editor, the compile job is restarted. This process can be repeated up to 9 times, depending on the single digit number specified in the option. If no number is specified in the option, this compile-edit-compile process repeats indefinitely until all errors are corrected. **–edit0** turns off this edit feature.

–trapuv

Force all un-initialized stack, automatic and dynamically allocated variables to be initialized with `0xFFFA5A5A`. When this value is used as a floating point variable, it is treated as a floating point NaN and it will cause a floating point trap. When it is used as a pointer, an address or segmentation violation will most likely occur.

−j Compile the specified source programs, and leave the *ucode* object file output in corresponding files suffixed with '.u'.

−ko *output*
Name the output file created by the ucode loader as *output*. This file is not removed. If this file is compiled, the object file is left in a file whose name consists of *output* with the suffix changed to a '.o'. If *output* has no suffix, a '.o' suffix is appended to *output*.

−k Pass options that start with a **−k** to the ucode loader. This option is used to specify ucode libraries (with **−kl** *x*) and other ucode loader options.

−G *num*
Specify the maximum size, in bytes, of a data item that is to be accessed from the global pointer. *num* is assumed to be a decimal number. If *num* is zero, no data is accessed from the global pointer. The default value for *num* is 8 bytes.

−std Have the compiler produce warnings for things that are not standard in the language.

−nocpp
Do not run the C macro preprocessor on C and assembly source files before compiling.

−signed
Cause all *char* declarations to be *signed char* declarations, the default is to treat them as *unsigned char* declarations.

−volatile
Causes all variables to be treated as *volatile*.

−varargs
Prints warnings for lines that may require the *varargs.h* macros.

−float Cause the compiler to never promote expressions of type *float* to type *double*.

The options described below primarily aid compiler development and are not generally used:

−H*c* Halt compiling after the pass specified by the character *c*, producing an intermediate file for the next pass. The *c* can be [**fjusmoca**]. It selects the compiler pass in the same way as the **−t** option. If this option is used, the symbol table file produced and used by the passes, is the last component of the source file with the suffix changed to '.T' and is not removed.

−K Build and use intermediate file names with the last component of the source file's name replacing its suffix with the conventional suffix for the type of file (for example '.B' file for binary *ucode*, produced by the front end). These intermediate files are never removed even when a pass encounters a fatal error. When ucode linking is performed and the **−K** option is specified the base name of the files created after the ucode link is 'u.out' by default. If **−ko** *output* is specified, the base name of the object file is *output* without the suffix if it exists or suffixes are appended to *output* if it has no suffix.

−# Converts binary *ucode* files ('.B') or optimized binary ucode files ('.O') to symbolic *ucode* (a '.U' file) using *btou*(1). If a symbolic ucode file is to be produced by converting the binary *ucode* from the C compiler front end then the front end option **−Xu** is used instead of *btou*(1).

−W*c*[*c*...],*arg1*[,*arg2*...]
Pass the argument[s] *argi* to the compiler pass[es] *c*[*c*..]. The *c*'s are one of [**pfjusmocablyz**]. The c's selects the compiler pass in the same way as the **−t** option.

The options **−t** [hpfjusmocablyzrnt], **−h** *path*, and **−B** *string* select a name to use for a particular pass, startup routine, or standard library. These arguments are processed from left to right so their order is significant. When the **−B** option is encountered, the selection of names takes place using the last **−h** and **−t** options. Therefore, the **−B** option is always required when using **−h** or **−t**. Sets of these options can be used to select any combination of names.

−t [hpfjusmocablyzrnt]

Select the names. The names selected are those designated by the characters following the **−t** option according to the following table:

Name	**Character**	
include	h	(see note below)
cpp	p	
ccom	f	
ujoin	j	
uld	u	
usplit	s	
umerge	m	
uopt	o	
ugen	c	
as0	a	
as1	b	
ld	l	
ftoc	y	
cord	z	
[m]crt[1n].o	r	
libprof1.a	n	
btou, utob	t	

If the character 'h' is in the **−t** argument then a directory is added to the list of directories to be used in searching for '#include' files. This directory name has the form /usr/include*string* . The standard directory is still searched.

−h*path*
Use *path* rather than the directory where the name is normally found.

−B*string*
Append *string* to all names specified by the **−t** option. If no **−t** option has been processed before the **−B**, the **−t** option is assumed to be "hpfjusmocablyzrnt". This list designates all names. If no **−t** argument has been processed before the **−B** then a **−B** *string* is passed to the loader to use with its **−l** *x* arguments.

If the environment variable TMPDIR is set, the value is used as the directory to place any temporary files rather than the default **/tmp/**.

FILES

file.o	Input file
file.o	Object file
a.out	Loaded output
/bin/cc	
/bin/as	
/bin/ld	
$COMP_HOST_ROOT/usr/lib/cmplrs/ld	The link editor start up program.
/lib/cpp	
$COMP_HOST_ROOT/usr/lib/cmplrs/oldc/cpp	The C-preprocessor.
/usr/include	Standard directory for '#include' files.
/usr/include2.20	Include directory for this version's '#include' files.
$COMP_HOST_ROOT/usr/lib/cmplrs/oldc/ccom	C front end.

$COMP_HOST_ROOT/usr/lib/cmplrs/ujoin
Binary ucode and symbol table joiner.

$COMP_HOST_ROOT/usr/lib/cmplrs/uld
Ucode loader.

$COMP_HOST_ROOT/usr/lib/cmplrs/usplit
Binary ucode and symbol table splitter

$COMP_HOST_ROOT/usr/lib/cmplrs/umerge
Procedure intergrator

$COMP_HOST_ROOT/usr/lib/cmplrs/uopt
Optional global ucode optimizer

$COMP_HOST_ROOT/usr/lib/cmplrs/ugen
Code generator

$COMP_HOST_ROOT/usr/lib/cmplrs/as0
Symbolic to binary assembly language translator.

$COMP_HOST_ROOT/usr/lib/cmplrs/as1
Binary assembly language assembler and reorganizer.

$COMP_HOST_ROOT/usr/lib/cmplrs/ld
The link editor.

$COMP_HOST_ROOT/usr/lib/cmplrs/btou
Binary to symbolic ucode translator

$COMP_HOST_ROOT/usr/lib/cmplrs/utob
Symbolic to binary ucode translator

/lib/libc.a Standard library.

/lib/crt1.o C runtime startup module.

SEE ALSO

as(1), cpp(1), ld(1).

"The C Programmer's Handbook" by M. I. Bolsky, Prentice-Hall and AT&T, 1985, ISBN 0-13-110073-4.

"The C Programming Language" by B. W. Kernighan and D. M. Ritchie, Prentice-Hall, 1978, ISBN 0-13-110163-3.

"Programming in C – A Tutorial" by B. W. Kernighan.

"C Reference Manual" by D. M. Ritchie.

"C Language" in the *Programming Guide.*

WARNING

By default, the return value from a C program is completely random. The only two guaranteed ways to return a specific value are to explicitly call *exit*(2) or to leave the function main(1) with a *return expression*; construct.

NOTICE

Profiling is not yet supported.

NAME

cdc − change the delta commentary of an SCCS delta

SYNOPSIS

cdc −**r**SID [−**m**[mrlist]] [−**y**[comment]] files

DESCRIPTION

cdc changes the *delta commentary*, for the **SID** (**S**CCS **ID**entification) string specified by the −**r** keyletter, of each named SCCS file.

delta commentary is defined to be the **M**odification **R**equest (**MR**) and comment information normally specified via the *delta*(1) command (−**m** and −**y** keyletters).

If a directory is named, *cdc* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of − is given, the standard input is read (see **WARNINGS**) and each line of the standard input is taken to be the name of an SCCS file to be processed.

Arguments to *cdc*, which may appear in any order, consist of *keyletter* arguments and file names.

All the described *keyletter* arguments apply independently to each named file:

−**r***SID* Used to specify the *S*CCS *ID*entification (SID) string of a delta for which the delta commentary is to be changed.

−**m***mrlist* If the SCCS file has the **v** flag set [see *admin*(1)] then a list of MR numbers to be added and/or deleted in the delta commentary of the SID specified by the −**r** keyletter *may* be supplied. A null MR list has no effect.

MR entries are added to the list of MRs in the same manner as that of *delta*(1). In order to delete an MR, precede the MR number with the character **!** (see EXAMPLES). If the MR to be deleted is currently in the list of MRs, it is removed and changed into a "comment" line. A list of all deleted MRs is placed in the comment section of the delta commentary and preceded by a comment line stating that they were deleted.

If **−m** is not used and the standard input is a terminal, the prompt **MRs?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The **MRs?** prompt always precedes the **comments?** prompt (see **−y** keyletter).

MRs in a list are separated by blanks and/or tab characters. An unescaped new-line character terminates the MR list.

Note that if the **v** flag has a value [see *admin*(1)], it is taken to be the name of a program (or shell procedure) which validates the correctness of the MR numbers. If a non-zero exit status is returned from the MR number validation program, *cdc* terminates and the delta commentary remains unchanged.

−y*[comment]* Arbitrary text used to replace the *comment*(s) already existing for the delta specified by the **−r** keyletter. The previous comments are kept and preceded by a comment line stating that they were changed. A null *comment* has no effect.

If **−y** is not specified and the standard input is a terminal, the prompt **comments?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped new-line character terminates the *comment* text.

Simply stated, the keyletter arguments are either (1) if you made the delta, you can change its delta commentary; or (2) if you own the file and directory you can modify the delta commentary.

EXAMPLES

```
cdc -r1.6 -m"bl78-12345 !bl77-54321 \
bl79-00001" -ytrouble s.file
```

adds bl78-12345 and bl79-00001 to the MR list, removes bl77-54321 from the MR list, and adds the comment **trouble** to delta 1.6 of s.file.

```
cdc -r1.6 s.file
MRs? !bl77-54321 bl78-12345 bl79-00001
comments? trouble
```

does the same thing.

WARNINGS

If SCCS file names are supplied to the *cdc* command via the standard input (− on the command line), then the **−m** and **−y** keyletters must also be used.

FILES

x-file	[see *delta*(1)]
z-file	[see *delta*(1)]

SEE ALSO

admin(1), delta(1), get(1), prs(1), sccsfile(4).

This page is intentionally left blank

NAME

cflow – generate C flowgraph

SYNOPSIS

cflow [−**r**] [−**ix**] [−**i_**] [−**d**num] files

DESCRIPTION

The *cflow* command analyzes a collection of C, yacc, lex, assembler, and object files and attempts to build a graph charting the external references. Files suffixed with **.y**, **.l**, and **.c** are yacced, lexed, and C-preprocessed as appropriate. The results of the preprocessed files, and files suffixed with **.i**, are then run through the first pass of *lint*(1). Files suffixed with **.s** are assembled. Assembled files, and files suffixed with **.o**, have information extracted from their symbol tables. The results are collected and turned into a graph of external references which is displayed upon the standard output.

Each line of output begins with a reference number, followed by a suitable number of tabs indicating the level, then the name of the global symbol followed by a colon and its definition. Normally only function names that do not begin with an underscore are listed (see the −**i** options below). For information extracted from C source, the definition consists of an abstract type declaration (e.g., **char** *), and, delimited by angle brackets, the name of the source file and the line number where the definition was found. Definitions extracted from object files indicate the file name and location counter under which the symbol appeared (e.g., *text*). Leading underscores in C-style external names are deleted.

Once a definition of a name has been printed, subsequent references to that name contain only the reference number of the line where the definition may be found. For undefined references, only < > is printed.

As an example, given the following in *file.c*:

```
int	i;

main()
{
	f();
	g();
	f();
}

f()
{
	i = h();
}
```

the command

```
cflow -ix file.c
```

produces the output

```
1	main: int(), <file.c 4>
2		f: int(), <file.c 11>
3			h: <>
4			i: int, <file.c 1>
5		g: <>
```

When the nesting level becomes too deep, the output of *cflow* can be piped to *pr*(1), using the **−e** option, to compress the tab expansion to something less than every eight spaces.

In addition to the **−D**, **−I**, and **−U** options [which are interpreted just as they are by *cc*(1) and *cpp*(1)], the following options are interpreted by *cflow*:

−r Reverse the "caller:callee" relationship producing an inverted listing showing the callers of each function. The listing is also sorted in lexicographical order by callee.

−ix Include external and static data symbols. The default is to include only functions in the flowgraph.

−i_ Include names that begin with an underscore. The default is to exclude these functions (and data if *−ix* is used).

−dnum The *num* decimal integer indicates the depth at which the flowgraph is cut off. By default this is a very large number. Attempts to set the cutoff depth to a nonpositive integer will be ignored.

DIAGNOSTICS

Complains about bad options. Complains about multiple definitions and only believes the first. Other messages may come from the various programs used (e.g., the C-preprocessor).

SEE ALSO

as(1), cc(1), cpp(1), lex(1), lint(1), nm(1), pr(1), yacc(1).

BUGS

Files produced by *lex*(1) and *yacc*(1) cause the reordering of line number declarations which can confuse *cflow*. To get proper results, feed *cflow* the *yacc* or *lex* input.

This page is intentionally left blank

NAME

clist – list C programs

SYNOPSIS

clist files

DESCRIPTION

clist produces a listing of the specified files on the standard output device.

clist will print 60 lines on a page. A line number is printed in front of each line. Each page has a heading containing the file name, the time of the last modification of the file, and the page number. The pages are numbered individually for each file.

A line containing only the characters of

/*$P*/

starting in column 1 will not be printed, instead the page will be ejected.

SEE ALSO

pr(1).

This page is intentionally left blank

NAME

comb – combine SCCS deltas

SYNOPSIS

comb [**−o**] [**−s**] [**−p** *sid*] [**−c** *list*] *files*

DESCRIPTION

comb generates a shell procedure [see *sh*(1)] which, when run, will reconstruct the given SCCS files. The reconstructed files will, hopefully, be smaller than the original files. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *comb* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored. The generated shell procedure is written on the standard output.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed, but the effects of any keyletter argument apply independently to each named file.

−o For each *get −e* generated, this argument causes the reconstructed file to be accessed at the release of the delta to be created, otherwise the reconstructed file would be accessed at the most recent ancestor. Use of the **−o** keyletter may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file.

−s This argument causes *comb* to generate a shell procedure which, when run, will produce a report giving, for each file: the file name, size (in blocks) after combining, original size (also in blocks), and percentage change computed by:

100 ∗ (original – combined) / original

It is recommended that before any SCCS files are actually combined, one should use this option to determine exactly how much space is saved by the combining process.

−pSID The *S*CCS *ID*entification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

−clist A *list* (see *get*(1) for the syntax of a *list*) of deltas to be preserved. All other deltas are discarded.

If no keyletter arguments are specified, *comb* will preserve only leaf deltas and the minimal number of ancestors needed to preserve the tree.

FILES

s.COMB	The name of the reconstructed SCCS file.
comb?????	Temporary.

SEE ALSO

admin(1), delta(1), get(1), help(1), prs(1), sh(1), sccsfile(4).

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

comb may rearrange the shape of the tree of deltas. It may not save any space; in fact, it is possible for the reconstructed file to actually be larger than the original.

NAME

cpp – the C language preprocessor

SYNOPSIS

/lib/cpp [option ...] [ifile [ofile]]

DESCRIPTION

cpp is the C language preprocessor which is invoked as the first pass of any C compilation using the *cc*(1) command. The output of *cpp* is designed to be in a form acceptable as input to the next pass of the C compiler. As the C language evolves, *cpp* and the rest of the C compilation package will be modified to follow these changes. Therefore, the use of *cpp* other than in this framework is not suggested. The preferred way to invoke *cpp* is through the *cc*(1) command since the functionality of *cpp* may someday be moved elsewhere. See *m4*(1) for a general macro processor.

cpp optionally accepts two filenames as arguments. *ifile* is the input and *ofile* is the output for the preprocessor. They default to standard input and standard output if not supplied.

The following *options* to *cpp* are recognized:

–V Print the version of *cpp*.

–P Preprocess the input without producing the line control information used by the next pass of the C compiler.

–C Pass along all comments except those found on *cpp* directive lines. By default, \f2cpp strips C-style comments.

–U*name*
: Remove any initial definition of *name* , where *name* is a reserved symbol that is predefined by the particular preprocessor.

–D*name*
–D*name*=*def*
: Define *name* as if by a **#define** directive. If no =*def* is given, *name* is defined as 1.

−I*dir* Change the algorithm for searching for **#include** files whose names do not begin with / to look in *dir* before looking in the directories on the standard list. When this option is used, **#include** files whose names are enclosed in "" are searched for first in the directory of the *ifile* argument, then in directories named in **−I** options, and last in directories on a standard list. For **#include** files whose names are enclosed in < >, **the directory of the** *ifile* argument is not searched.

−a No white spaces allowed before # in *cpp* directives (see below).

−p Don't replace **#param** in replacement-strings.

−d4 Print the number of bytes allocated by *cpp*.

−E Ignored

−v Ignored

Four special names are understood by *cpp*. The name **_LINE_** is defined as the current line number (as a decimal integer) as known by *cpp*, **_FILE_** is defined as the current filename (as a C string) as known by *cpp*, **_DATE_** is defined as the current date (as a C-string), and **_TIME_** is defined as the current time (as a C-string in the form "hh:mm:ss"). They can be used anywhere (including in macros) just as any other defined name.

All *cpp* directives start with lines begun by # optionally after white spaces. The directives are:

#define *name token-string*
Replace subsequent instances of *name* with *token-string*.

#define *name***(** *arg***, ...,** *arg* **)** *token-string*
Notice that there can be no space between *name* and the **(**. Replace subsequent instances of *name* followed by a **(**, a list of comma-separated tokens, and a **)** by *token-string* where each occurrence of an *arg* in the *token-string* is replaced by the corresponding token in the comma-separated list.

#undef *name*

Cause the definition of *name* (if any) to be forgotten from now on.

#include "filename"
#include <*filename*>

Include at this point the contents of *filename* (which will then be run through *cpp*). When the <*filename*> notation is used, *filename* is only searched for in the standard places. See the **−I** option above for more detail.

#line *integer-constant "filename"*

Causes *cpp* to generate line control information for the next pass of the C compiler. *integer-constant* is the line number of the next line and *filename* is the file where it comes from. If *"filename"* is not given, the current filename is unchanged.

#error *info*

Causes *cpp* to generate a message including *info* on standard output.

#pragma *info*

Causes *cpp* to generate '#p info'.

\# The empty cpp-directive has no effect.

#endif

Ends a section of lines begun by a test directive (**#if**, **#ifdef**, or **#ifndef**). Each test directive must have a matching **#endif**.

#ifdef *name*

The lines following appear in the output if and only if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

#ifndef *name*

The lines following do not appear in the output if and only if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

#if *constant-expression*

Lines following appear in the output if and only if the *constant-expression* evaluates to non-zero. All binary non-assignment C operators, the **?:** operator, the unary −, **!**, and ~ operators are all legal in *constant-expression*. The precedence of the operators is the same as defined by the C language. There is also a unary operator **defined**, which can be used in *constant-expression* in these two forms: **defined** (*name*) or **defined** *name*. This allows the utility of **#ifdef** and **#ifndef** in a **#if** directive. Only these operators, integer constants, and names which are known by *cpp* should be used in *constant-expression*. In particular, the **sizeof** operator is not available.

#else

Reverses the notion of the test directive that matches this directive. If lines previous to this directive are ignored, the following lines appear in the output. If lines previous to this directive are not ignored, the following lines do not appear in the output.

#elif

Like **#else #if** but does not need an **#endif**.

The test directives and the possible **#else** directives can be nested.

FILES

/usr/include standard directory for **#include** files

SEE ALSO

cc(1), m4(1).

DIAGNOSTICS

The error messages produced by *cpp* are self-explanatory. The line number and filename where the error occurred are printed along with the diagnostic.

NOTES

When newline characters were found in argument lists for macros to be expanded, previous versions of *cpp* put out the newlines as they were found and expanded. The current version of *cpp* replaces these newlines with blanks to alleviate problems that the previous versions had when this occurred.

Unlike when using **cc** command the __STDC__ macro is not default defined for /lib/cpp.

This page is intentionally left blank

NAME

cprs – compress a common object file

SYNOPSIS

cprs [**–p**] file1 file2

DESCRIPTION

The *cprs* command reduces the size of a common object file, *file1*, by removing duplicate structure and union descriptors. The reduced file, *file2*, is produced as output.

The sole option to *cprs* is:

–p Print statistical messages including: total number of tags, total duplicate tags, and total reduction of *file1*.

SEE ALSO

strip(1), a.out(4), syms(4).

This page is intentionally left blank

NAME

ctrace – C program debugger

SYNOPSIS

ctrace [options] [file]

DESCRIPTION

The *ctrace* command allows you to follow the execution of a C program, statement-by-statement. The effect is similar to executing a shell procedure with the **–x** option. *ctrace* reads the C program in *file* (or from standard input if you do not specify *file*), inserts statements to print the text of each executable statement and the values of all variables referenced or modified, and writes the modified program to the standard output. You must put the output of *ctrace* into a temporary file because the *cc*(1) command does not allow the use of a pipe. You then compile and execute this file.

As each statement in the program executes it will be listed at the terminal, followed by the name and value of any variables referenced or modified in the statement, followed by any output from the statement. Loops in the trace output are detected and tracing is stopped until the loop is exited or a different sequence of statements within the loop is executed. A warning message is printed every 1000 times through the loop to help you detect infinite loops. The trace output goes to the standard output so you can put it into a file for examination with an editor or the *bfs*(1) or *tail*(1) commands.

The options commonly used are:

–f *functions*	Trace only these *functions*
–v *functions*	Trace all but these *functions*

You may want to add to the default formats for printing variables. Long and pointer variables are always printed as signed integers. Pointers to character arrays are also printed as strings if appropriate. Char, short, and int variables are also printed as signed integers and, if appropriate, as characters. Double variables are printed as floating point numbers in scientific notation. You can request that variables be printed in additional formats, if appropriate, with these options:

−o Octal
−x Hexadecimal
−u Unsigned
−e Floating point

These options are used only in special circumstances:

−l *n* Check *n* consecutively executed statements for looping trace output, instead of the default of 20. Use 0 to get all the trace output from loops.

−s Suppress redundant trace output from simple assignment statements and string copy function calls. This option can hide a bug caused by use of the = operator in place of the = = operator.

−t *n* Trace *n* variables per statement instead of the default of 10 (the maximum number is 20). The Diagnostics section explains when to use this option.

−P Run the C preprocessor on the input before tracing it. You can also use the **−D**, **−I**, and **−U** *cpp*(1) options.

These options are used to tailor the run-time trace package when the traced program will run in a non-UNIX System environment:

−b Use only basic functions in the trace code, that is, those in *ctype*(3C), *printf*(3S), and *string*(3C). These are usually available even in cross-compilers for microprocessors. In particular, this option is needed when the traced program runs under an operating system that does not have *signal*(2), *fflush*(3S), *longjmp*(3C), or *setjmp*(3C).

−p *string*
Change the trace print function from the default of 'printf('. For example, 'fprintf(stderr,' would send the trace to the standard error output.

−r *f* Use file *f* in place of the *runtime.c* trace function package. This lets you change the entire print function, instead of just the name and leading arguments (see the **−p** option).

EXAMPLE

If the file *lc.c* contains this C program:

```
#include <stdio.h>
main()/* count lines in input */
{
int c, nl;

nl = 0;
while ((c = getchar()) != EOF)
if (c = '\n')
++nl;
printf("%d\n", nl);
}
```

and you enter these commands and test data:

```
cc lc.c
a.out
1
(cntl-d)
```

the program will be compiled and executed. The output of the program will be the number **2**, which is not correct because there is only one line in the test data. The error in this program is common, but subtle. If you invoke *ctrace* with these commands:

```
ctrace lc.c >temp.c
cc temp.c
a.out
```

the output will be:

```
2 main()
6 nl = 0;
  /* nl == 0 */
7 while ((c = getchar()) != EOF)
```

The program is now waiting for input. If you enter the same test data as before, the output will be:

```
    /* c == 49 or '1' */
8 if (c = '\n')
    /* c == 10 or '\n' */
9 ++nl;
    /* nl == 1 */
7 while ((c = getchar()) != EOF)
    /* c == 10 or '\n' */
8 if (c = '\n')
    /* c == 10 or '\n' */
9 ++nl;
    /* nl == 2 */
7 while ((c = getchar()) != EOF)
```

If you now enter an end of file character (cntl-d) the final output will be:

```
    /* c == -1 */
10 printf("%d\n", nl);
    /* nl == 2 */2
return
```

Note that the program output printed at the end of the trace line for the **nl** variable. Also note the **return** comment added by *ctrace* at the end of the trace output. This shows the implicit return at the terminating brace in the function.

The trace output shows that variable **c** is assigned the value '1' in line 7, but in line 8 it has the value '\n'. Once your attention is drawn to this **if** statement, you will probably realize that you used the assignment operator (=) in place of the equality operator (==). You can easily miss this error during code reading.

EXECUTION-TIME TRACE CONTROL

The default operation for *ctrace* is to trace the entire program file, unless you use the **−f** or **−v** options to trace specific functions. This does not give you statement-by-statement control of the tracing, nor does it let you turn the tracing off and on when executing the traced program.

You can do both of these by adding *ctroff*() and *ctron*() function calls to your program to turn the tracing off and on, respectively, at execution time. Thus, you can code arbitrarily complex criteria for trace control with *if* statements, and you can even conditionally include this code because *ctrace* defines the **CTRACE** preprocessor variable. For example:

```
#ifdef CTRACE
if (c == '!' && i > 1000)
ctron();
#endif
```

You can also call these functions from *sdb*(1) if you compile with the **−g** option. For example, to trace all but lines 7 to 10 in the main function, enter:

```
sdb a.out
main:7b ctroff()
main:11b ctron()
r
```

You can also turn the trace off and on by setting static variable tr_ct_ to 0 and 1, respectively. This is useful if you are using a debugger that cannot call these functions directly.

DIAGNOSTICS

This section contains diagnostic messages from both *ctrace* and *cc*(1), since the traced code often gets some *cc* warning messages. You can get *cc* error messages in some rare cases, all of which can be avoided.

ctrace Diagnostics

warning: some variables are not traced in this statement

Only 10 variables are traced in a statement to prevent the C compiler "out of tree space; simplify expression" error. Use the **−t** option to increase this number.

warning: statement too long to trace

This statement is over 400 characters long. Make sure that you are using tabs to indent your code, not spaces.

cannot handle preprocessor code, use −P option

This is usually caused by #ifdef/#endif preprocessor statements in the middle of a C statement, or by a semicolon at the end of a #define preprocessor statement.

'if ... else if' sequence too long

Split the sequence by removing an **else** from the middle.

possible syntax error, try −P option

Use the **−P** option to preprocess the *ctrace* input, along with any appropriate **−D**, **−I**, and **−U** preprocessor options. If you still get the error message, check the Warnings section below.

Cc Diagnostics

warning: illegal combination of pointer and integer
warning: statement not reached
warning: sizeof returns 0

Ignore these messages.

compiler takes size of function

See the *ctrace* "possible syntax error" message above.

yacc stack overflow

See the *ctrace* "'if ... else if' sequence too long" message above.

out of tree space; simplify expression

Use the **−t** option to reduce the number of traced variables per statement from the default of 10. Ignore the "ctrace: too many variables to trace" warnings you will now get.

redeclaration of signal

Either correct this declaration of *signal*(2), or remove it and #include <signal.h>.

SEE ALSO

bfs(1), tail(1), signal(2), ctype(3C), setjmp(3C), string(3C), fclose(3S), printf(3S).

WARNINGS

You will get a *ctrace* syntax error if you omit the semicolon at the end of the last element declaration in a structure or union, just before the right brace (}). This is optional in some C compilers.

Defining a function with the same name as a system function may cause a syntax error if the number of arguments is changed. Just use a different name.

ctrace assumes that BADMAG is a preprocessor macro, and that EOF and NULL are #defined constants. Declaring any of these to be variables, e.g., "int EOF;", will cause a syntax error.

BUGS

ctrace does not know about the components of aggregates like structures, unions, and arrays. It cannot choose a format to print all the components of an aggregate when an assignment is made to the entire aggregate. *ctrace* may choose to print the address of an aggregate or use the wrong format (e.g., 3.149050e-311 for a structure with two integer members) when printing the value of an aggregate.

Pointer values are always treated as pointers to character strings.

The loop trace output elimination is done separately for each file of a multi-file program. This can result in functions called from a loop still being traced, or the elimination of trace output from one function in a file until another in the same file is called.

FILES

/usr/lib/ctrace/runtime.crun-time trace package

This page is intentionally left blank

NAME

cxref – generate C program cross-reference

SYNOPSIS

cxref [options] files

DESCRIPTION

The *cxref* command analyzes a collection of C files and attempts to build a cross-reference table. *cxref* uses a special version of *cpp* to include **#define**'d information in its symbol table. It produces a listing on standard output of all symbols (auto, static, and global) in each file separately, or, with the **−c** option, in combination. Each symbol contains an asterisk (*) before the declaring reference.

In addition to the **−D**, **−I** and **−U** options [which are interpreted just as they are by *cc*(1) and *cpp*(1)], the following *options* are interpreted by *cxref*:

−c Print a combined cross-reference of all input files.

−w<*num*>
Width option which formats output no wider than <num> (decimal) columns. This option will default to 80 if <num> is not specified or is less than 51.

−o file Direct output to *file*.

−s Operate silently; do not print input file names.

−t Format listing for 80-column width.

FILES

LLIBDIR usually /usr/lib

LLIBDIR/xcpp special version of the C preprocessor.

SEE ALSO

cc(1), cpp(1).

DIAGNOSTICS

Error messages are unusually cryptic, but usually mean that you cannot compile these files.

BUGS

cxref considers a formal argument in a *#define* macro definition to be a declaration of that symbol. For example, a program that *#includes* **ctype.h**, will contain many declarations of the variable **c**.

NAME

dbx – source-level debugger

SYNOPSIS

dbx [**–I** *directory*] [**–c** *file*] [–i] [–r] [*object*] [*core*]

DESCRIPTION

dbx is a source-level debugger for the Supermax RISC.

The object file used with the debugger is produced by specifying an appropriate option (**–g**) to the compiler. The resulting object file contains symbol table information, including the names of all source files that the compiler translated to create the object file. These source files are accessible from the debugger. If **–g** is not specified, limited debugging is possible.

If a core file exists in the current directory or a coredump is specified, *dbx* can be used to look at the state of the program when it faulted. *dbx* does not support lines greater than 511.

Running dbx

If a *.dbxinit* file resides in the current directory or in the user's home directory, the commands in it are executed when *dbx* is invoked.

When invoked, *dbx* recognizes these command line options:

–I *directory or* **–I***directory*
: Tells *dbx* to look in the specified directory for source files. Multiple directories can be specified by using multiple *–I* options. *dbx* searches for source files in the current directory and in the object file's directory whether or not *–I* is used.

–c *file*
: Selects a command file other than *.dbxinit*.

–i
: Uses interactive mode. This option does not treat #s as comments in a file. It prompts for source even when it reads from a file. With this option, *dbx* also has extra formatting as if for a terminal.

−r Runs the object file immediately.

The *dbx* monitor offers powerful command line editing. For a full description of these editing features, see *csh*(1).

Multiple commands can be specified on the same command line by separating them with a semicolon (;). If the user types a string and presses the stop character usually (^z; see *stty*(1), *dbx* tries to complete a symbol name from the program that matches the string.

The Monitor

These commands control the *dbx* monitor:

![*string*] [*integer*] [−*integer*]
: Specifies a command from the history list.

help Prints a list of *dbx* commands, using the UNIX system 'more' command to display the list.

history Prints the items from the history list. The default is 20.

quit[!] Exit dbx after verification. If '!' is specified, verification is not required.

Controlling dbx

alias [*name*(*arg1,...argN*)"*string*"]
: Lists all existing aliases, or, if an argument is specified, defines a new alias.

unalias *alias command name*
: Removes the specified alias.

delete *expression1,...expressionN*

delete *all* Deletes the specified item from the status list. The argument *all* deletes all items from the status list.

playback input [*file*]
: Replays commands that were saved with the record input commands in a text file.

playback output [*file*]
: Replays debugger output that was saved with the record output command.

record input [*file*]
: Records all commands typed to dbx.

record output [*file*]
: Records all dbx output.

sh [*shell command*]
: Calls a shell from dbx or executes a shell command.

status
: Lists currently set stop, record, and trace commands.

tagvalue (*tagname*)
: Returns the value of *tagname*. If the tags extends to more than one line, or if it contains arguments, an error occurs. *tagvalue* can be used in any expression.

set [*variable* = *expression*]
: Lists existing debugger variables and their values. This command can also be used to assign a new value to an existing variable or to define a new variable.

unset *variable*
: Removes the setting of a specified debugger variable.

Examining Source

/regular expression
: Searches ahead in the source code for the regular expression.

?regular expression
: Searches back in the source code for the regular expression.

edit [*file*] Calls an editor from dbx.

file [*file*] Prints the current file name, or, if a file name is specified, this command changes the current file to the specified file.

func [*expression*] [*procedure*]
Moves to the specified procedure (activation level), or, if an expression or procedure is not specified, prints the current activation level.

list [*expression:integer*]

list [*expression*]
Lists the specified lines. The default is 10 lines.

tag *tagname* Sets the current file/line to the location specified by *tagname*. Operations are similar to tge tag operations in *vi*(1).

use [*directory1 . . . directoryN*]
Lists source directories, or, if a directory name is specified, this command substitutes the new directories for the previous list.

whatis *variable*
Prints the type declaration for the specified name.

which *variable*
Finds the variable name currently being used.

whereis *variable*
Prints all qualifications (the scopes) of the specified variable name.

Controlling Programs

assign *expression1* = *expression2*
Assigns the specified expression to a specified program variable.

[*n*] **cont** [*signal*]

cont [*signal*] **to** *line*

cont [*signal*] **in** *procedure*

Continues executing a program after a breakpoint. *n* breakpoints are ignored if *n* is specified before stepping. IOIf specified, *signal* is delivered to the processing being debugged.

goto *line* — Goes to the specified line in the source.

next [*integer*] — Steps over the specified number of lines. The default is one. This command does not step into procedures.

rerun [*arg1* ... *argN*] [<*file1*] [>*file2*]

rerun [*arg1* ... *argN*] [<*file1*] [>&*file2*]

Reruns the program, using the same arguments that were specified to the run command. If new arguments are specified, rerun uses those arguments.

run [*arg1* ... *argN*] [<*file1*] [>*file2*]

run [*arg1* ... *argN*] [<*file1*] [>&*file2*]

Runs the program with the specified arguments.

return [*procedure*]

Continues executing until the procedire returns. If a procedure is not specified, dbx assumes the next procedure.

step [*integer*] — Steps the specified number of lines. This command steps into procedures. The default is one line.

Setting Breakpoints

catch [*signal*]

Lists al signals that dbx catches, or, if an argument is specified, adds a new signal to the catch list.

ignore [*signal*]
Lists all signals that dbx does not catch. If a signal is specified, this command adds the signal to the ignore list.

stop [*variable*]

stop [*variable*] **at** *line* [**if** *expression*]

stop [*variable*] **in** *procedure* [**if** *expression*]

stop [*variable*] **if** *expression*
Sets the breakpoint at the specified point.

trace *variable* [**at** *line*] [*if expression*]

trace *variable* [**in** *procedure*] [*if expression*]
Traces the specified variable.

when [*variable*] [**at** *line*] {*command list*}

when [*variable*] [**in** *procedure*] {*command list*}
Executes the specified dbx comma separated command list.

Examining Program State

dump [*procedure*] [.]
Prints variable information about procedure. If a dot (.) is specified, this command prints global variable information on all procedures in the stack and the variables of those procedures.

down [*expression*]
Moves down the specified number of activation levels in the stack. The default is one level.

up [*expression*]
Moves up the specified number of activation levels in the stack. The default is one level.

print *expression1, ... expressionN*
Prints the value of the specified expression. If *expression* is a dbx keyword, it must be enclosed in parantheses. For example, to print out a variable called 'output' (which is also a

variable in the playback and record commands) you must type:

print (output)

printf *"string", expression1, ... expressionN*
Prints the value of the specified expression, using C language string formatting. As in the print command, if *expression* is a dbx keyword, you must enclose it within parantheses.

printregs Prints all register values.

where Does a stack trace, which shows the current activation levels.

where *n* Prints out only the top *n* levels of the stack.

Debugging at the Machine Level

[*n*] **conti** [*signal*]

conti [*signal*] **to** *address*

conti [*signal*] **in** *peocedure*
Continues executing assembly code after a breakpoint. *n* breakpoints are ignored if *n* is specified before stepping. If specified, *signal* is delivered to the processing being debugged.

nexti [*integer*]
Steps over the specified number of machine instructions. The default is one. This command does not step into procedures.

stepi [*integer*]
Steps the specified number of machine instructions. This command steps into procedures. The default is one instruction.

stopi [*variable*] **at** [*address*] [**at** *address* [**if** *expression*]

stopi [*variable*] **in** *procedure* [**if** *expression*]

stopi [*variable*] **if** *expression*

Sets the breakpoint in the machine code at the specified point.

tracei *variable* **at** *address* [**at** *address* **if** *expression*]

tracei *variable* **in** *procedure* [**at** *address* **if** *expression*]

Traces the specified variable in machcine instructions.

wheni [*variable*] [**at** *address*] {*command*}

wheni [*variable*] [**in** *procedure*] {*command*}

Executes the specified dbx comma separated command list.

address[?]/ <**count**> <**mode**>

Searching forward (or backward, if **?** is specified), prints the contents *address*, or disassembles the code for the instruction *address*; *count* is the number of items to be printed at the specified address. *mode* is one of the characters in the following table producing the indicated result:

d	Print a short word in decimal.
D	Print a long word in decimal.
o	Print a short word in octal.
O	Print a long word in octal.
x	Print a short word in hexadecimal.
X	Print a long word in hexadecimal.
b	Print a byte in octal.
c	Print a bite as a character.
s	Print a string of characters that ends in a null.
f	Print a single precision real number.
g	Print a double precision real number.
i	Print machine instructions.
n	Prints data in typed format.

address / < *countL* > < *value* > < *mask* >
: Searches for a 32-bit word starting at the specified *address*; *count* specifies the number of word to process in the search; an address is printed when the word *at address*, after an AND operation with *mask*, is equal to *value*.

Predefined dbx Variables:

The debugger has these predefined variables:

$addfmt	Specifies the format for addresses. This can be set to any specification that a C 'printf' statement can format. The default is zero.
$byteaccess	Same as **$addrfmt**.
$casesence	When set to a nonzero value, specifies that uppercase and lowercase letters be taken into consideration during a search. When set to 0, the case is ignored. The default i 0.
$curevent	Shows the last even number as seen in the status feature. Set only by dbx.
$curline	Specifies the current line. Set only by dbx.
$curscrline	Shows the last line listed plus 1. Set only by dbx.
$curpc	Specifies the current address. Used with the *wi* and *li* aliases.
$datacache	Caches information from the data space so that dbx must access data space only once. To debug the operating system, set this variable to 0; otherwise set it to a nonzero value. The default is 1.
$debugflag	For internal use by dbx.
$defin	For internal use by dbx.

$defout For internal use by dbx.

$dispix For use when debugging pixie code. When set to 0, machine code is showed while debugging. When set to 1, pixie code is shown. The default is 0.

$hexchars Output characters are printed in hexadecimal format (set, unset).

$hexin Specifies that inout constants are hexadecimal.

$hexints When set to a nonzero value, changes the default output constants toi hexadecimal. Overrides *$octints*.

$hexstrings When set to 1, specifies that all strings are printed in hexadecimal; when set to 0, strings are printed in character format.

$historyevent
Shows the current history line.

$lines Number of lines for history. The default is 20.

$listwindow Specifies how many lines the *list* command prints.

$main Specifies the name of the procedure that dbx will start with. This can be set to any procedure. The default is 'main'.

$maxstrlen Specifies how many characters of a string that dbx prints for pointers to strings. The default is 128.

$octin When set to nonzero, changes the default input constants to octal. When set, *$hexint* overrides this setting.

$octints Output integers are printed octal format (set, unset).

$page Specifies whether to page long information. A nonzero value turns on paging; a 0 turns it off. The default is 1.

$pagewindow
Specifies how many lines print when information runs longer than one screen. This can be changed to match the number of lines on any terminal. If set to 0, this variable assumes one line. The default is 22, leaving space for continuation query.

$pdbxport Port name from */etc/remote*[*.pdbx*] used to connect to target machine for pdbx.

$printwhilestep
For use with the step[*n*] and stepi[*n*] instructions. A nonzero integer specifies that all *n* lines and/or instructions should be printed out. A zero specifies that only the last line and/or instruction should be printed out. The default is zero.

$pimode Prints input when used with the *playback input* command. The default is 0.

$printdata When set to a nonzero value, the contents of registers used are printed next to each instruction displayed. The default is 0.

$printwide When set to a nonzero value, the contents of variables are printed in a horizontal format. The default is 0.

$prompt Sets the prompt for dbx.

$readtextfile When set to 1, dbx tries to read instructions from the object file rather than the process. dbx executes faster when debugging remotely using the System Programmer's Package. This variable should always be set to 0 when the process being debugged copies in code during the debugging process. The default is 1.

$regstyle A zero value causes registers to be printed out in their normal *r* format (*r0,r1, ... r31*). A nonzero value causes the registers to be printed out in a special format (*zero, at, v0, v1, ...*) commonly used in debugging programs written in assembly language.

$repeatmode
When set to a nonzero value, after pressing the RETURN key (for an empty line), the last command is repeated. The default is 1.

$rimode When set to a nonzero value, input will is ???? recorded while recording output. The default is 0.

$sigtramp Tells dbx the name of the code called by the system to invoke user signal handlers. This variable is set to *sigtramp* system running under RISC/os.

$tagfile Contains a filename, indicating the file in which the tag command and the tabvalue macro are to search for tags.

Predefined dbx Aliases

The debugger has these predefined aliases:

? Prints a list of all dbx commands.

a Assigns a value to a program variable.

b Sets a breakpoint at a specified line.

bp Stops in a specified procedure.

c Continues program execution after a breakpoint.

d Deletes the specified item from the status list.

e Looks at the specified line.

f Moves to the specified activation level on the stack.

g Goes to the specified line and begins executing the program there.

h Lists all items currently on the history list.

j Shows what items are on the status list.

l Lists the next 10 lines of source code.

li Lists the next 10 machine instructions.

n or **S** Step over the specified number of lines without stepping into procedure calls.

ni or **Si**
Step over the specified number of assembly code instructions without stepping into procedure calls.

p Prints the value of the specified expression or variable.

pd Prints the value of the specified expression or variable in decimal.

pi Replays dbx commands that were saved with the record input format.

po Prints the value of the specified expression or variable in octal.

pr Prints values for all registers.

px Prints the value for the specified variable or expression in hexadecimal.

q Ends the debugging session.

r Runs the program again with the same arguments that were specified with the 'run' command.

ri Records in a file every command typed.

ro Records all debugger output in the specified file.

s Steps the next number of specified lines.

si Steps the next number of specified lines of assembly code instructions.

t Does a stack trace.

u Lists the previous 10 lines.

w Lists the 5 lines preceding and following the current line.

W Lists the 10 lines preceding and following the current line.

wi Lists the 5 machine instructions preceding and following the machine instruction.

NOTE:

In order to use all facilities in *dbx* it is important that the word **LINEEDIT=** is placed in the environment:

```
LINEEDIT=
export LINEEDIT
```

SEE ALSO

dbx in the *Programmers Guide.*

NAME

delta – make a delta (change) to an SCCS file

SYNOPSIS

delta [–**r**SID] [–**s**] [–**n**] [–**g**list] [–**m**[mrlist]] [–**y**[comment]] [–**p**] files

DESCRIPTION

delta is used to permanently introduce into the named SCCS file changes that were made to the file retrieved by *get*(1) (called the *g-file*, or generated file).

delta makes a delta to each named SCCS file. If a directory is named, *delta* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read (see *WARNINGS*); each line of the standard input is taken to be the name of an SCCS file to be processed.

delta may issue prompts on the standard output depending upon certain keyletters specified and flags [see *admin*(1)] that may be present in the SCCS file (see –**m** and –**y** keyletters below).

Keyletter arguments apply independently to each named file.

–**r***SID* Uniquely identifies which delta is to be made to the SCCS file. The use of this keyletter is necessary only if two or more outstanding *get*s for editing (**get** –**e**) on the same SCCS file were done by the same person (login name). The SID value specified with the –**r** keyletter can be either the SID specified on the *get* command line or the SID to be made as reported by the *get* command [see *get*(1)]. A diagnostic results if the specified SID is ambiguous, or, if necessary and omitted on the command line.

−s Suppresses the issue, on the standard output, of the created delta's SID, as well as the number of lines inserted, deleted and unchanged in the SCCS file.

−n Specifies retention of the edited *g-file* (normally removed at completion of delta processing).

−g*list* a *list* (see *get*(1) for the definition of *list*) of deltas which are to be *ignored* when the file is accessed at the change level (SID) created by this delta.

−m*[mrlist]*

If the SCCS file has the **v** flag set [see *admin*(1)] then a Modification Request (**MR**) number *must* be supplied as the reason for creating the new delta.

If **−m** is not used and the standard input is a terminal, the prompt **MRs?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The **MRs?** prompt always precedes the **comments?** prompt (see **−y** keyletter).

MRs in a list are separated by blanks and/or tab characters. An unescaped new-line character terminates the **MR** list.

Note that if the **v** flag has a value [see *admin*(1)], it is taken to be the name of a program (or shell procedure) which will validate the correctness of the **MR** numbers. If a non-zero exit status is returned from the **MR** number validation program, *delta* terminates. (It is assumed that the **MR** numbers were not all valid.)

−y[*comment]*
: Arbitrary text used to describe the reason for making the delta. A null string is considered a valid *comment*.

  If **−y** is not specified and the standard input is a terminal, the prompt **comments?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped new-line character terminates the comment text.

−p
: Causes *delta* to print (on the standard output) the SCCS file differences before and after the delta is applied in a *diff*(1) format.

FILES

g-file	Existed before the execution of *delta*; removed after completion of *delta*.
p-file	Existed before the execution of *delta*; may exist after completion of *delta*.
q-file	Created during the execution of *delta*; removed after completion of *delta*.
x-file	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z-file	Created during the execution of *delta*; removed during the execution of *delta*.
d-file	Created during the execution of *delta*; removed after completion of *delta*.
/usr/bin/bdiff	Program to compute differences between the "gotten" file and the *g-file*.

WARNINGS

Lines beginning with an **SOH** ASCII character (binary 001) cannot be placed in the SCCS file unless the **SOH** is escaped. This character has special meaning to SCCS [see *sccsfile*(4) (5)] and will cause an error.

A *get* of many SCCS files, followed by a *delta* of those files, should be avoided when the *get* generates a large amount of data. Instead, multiple *get/delta* sequences should be used.

If the standard input (−) is specified on the *delta* command line, the **−m** (if necessary) and **−y** keyletters *must* also be present. Omission of these keyletters causes an error to occur.

Comments are limited to text strings of at most 512 characters.

SEE ALSO

admin(1), bdiff(1), cdc(1), get(1), help(1), prs(1), rmdel(1), sccsfile(4).

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

dis – disassemble an object file.

SYNOPSIS

dis [options] files

DESCRIPTION

dis disassembles object files into machine instructions. Please note that assembler code and machine code can differ on this machine.

–h	Print the general register names rather than the software register names.
–p *procedure*	Disassembles only the specified procedure from the object file.
–S	Causes source lisitings to be listed. Otherwise, only instructions are listed.

SEE ALSO

as (1), cc (1), ld (1).

DIAGNOSTICS

The self-explanatory diagnostics indicate errors in the command line or problems encountered with the specified files.

This page is intentionally left blank

NAME

dump – dump selected parts of an object file

SYNOPSIS

dump [options] files

DESCRIPTION

The *pdump* command dumps selected parts of each of its object *file* arguments.

This command will accept both object files and archives of object files, but will only accept files of the same code-type, i.e code generated with the same values of the TARGETMC environment – see *intro*(1).

It processes each file argument according to one or more of the following options:

–a Dump the archive header of each member of each archive file argument.

–g Dump the global symbols in the symbol table of an archive.

–f Dump each file header.

–o Dump each optional header.

–h Dump section headers.

–s Dump section contents.

–r Dump relocation information.

–l Dump line number information.

–t Dump symbol table entries.

–z name Dump line number entries for the named function.

–c Dump the string table.

–L Interpret and print the contents of the *.lib* sections.

The following *modifiers* are used in conjunction with the options listed above to modify their capabilities.

−d number — Dump the section number, *number*, or the range of sections starting at *number* and ending at the *number* specified by **+d**.

+d number — Dump sections in the range either beginning with first section or beginning with section specified by **−d**.

−n name — Dump information pertaining only to the named entity. This *modifier* applies to **−h**, **−s**, **−r**, **−l**, and **−t**.

−p — Suppress printing of the headers.

−t index — Dump only the indexed symbol table entry. The **−t** used in conjunction with **+t**, specifies a range of symbol table entries.

+t index — Dump the symbol table entries in the range ending with the indexed entry. The range begins at the first symbol table entry or at the entry specified by the **−t** option.

−u — Underline the name of the file for emphasis.

−v — Dump information in symbolic representation rather than numeric (e.g., C_STATIC instead of **0X02**). This *modifier* can be used with all the above options except **−s** and **−o** options of *pdump*.

−z name,number

Dump line number entry or range of line numbers starting at *number* for the named function.

+z number — Dump line numbers starting at either function *name* or number specified by **−z**, up to *number* specified by **+z**.

−i Dumps the symbolic information header.

−F Dump the file descriptor table.

−P Dump the procedure descriptor table.

−R Dump the relative file index table.

Blanks separating an *option* and its *modifier* are optional. The comma separating the name from the number modifying the **−z** option may be replaced by a blank.

The *dump* command attempts to format the information it dumps in a meaningful way, printing certain information in character, hex, octal or decimal representation as appropriate.

SEE ALSO

a.out(4), ar(4).

This page is intentionally left blank

NAME

get – get a version of an SCCS file

SYNOPSIS

get [–**r**SID] [–**c**cutoff] [–**i**list] [–**x**list] [–**w**string] [–**a**seq-no.] [–**k**] [–**e**] [–**l**[**p**] [–**p**] [–**m**] [–**n**] [–**s**] [–**b**] [–**g**] [–**t**] file ...

DESCRIPTION

get generates an ASCII text file from each named SCCS file according to the specifications given by its keyletter arguments, which begin with –. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *get* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The generated text is normally written into a file called the *g-file* whose name is derived from the SCCS file name by simply removing the leading **s.**; (see also *FILES*, below).

Each of the keyletter arguments is explained below as though only one SCCS file is to be processed, but the effects of any keyletter argument applies independently to each named file.

–**r***SID* The *S*CCS *ID*entification string (SID) of the version (delta) of an SCCS file to be retrieved. Table˜1 below shows, for the most useful cases, what version of an SCCS file is retrieved (as well as the SID of the version to be eventually created by *delta*(1) if the –**e** keyletter is also used), as a function of the SID specified.

−c*cutoff* *Cutoff* date-time, in the form:

YY[MM[DD[HH[MM[SS]]]]]

No changes (deltas) to the SCCS file which were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, **−c7502** is equivalent to **−c750228235959**. Any number of non-numeric characters may separate the various 2-digit pieces of the *cutoff* date-time. This feature allows one to specify a *cutoff* date in the form: "**−c77/2/2 9:22:25**". Note that this implies that one may use the %E% and %U% identification keywords (see below) for nested *gets* within, say the input to a *send*(1C) command:

~!get "−c%E% %U%" s.file

−i*list* A *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

<list> ::= <range> | <list> , <range>
<range> ::= SID | SID − SID

SID, the SCCS Identification of a delta, may be in any form shown in the "SID Specified" column of Table 1.

−x*list* A *list* of deltas to be excluded in the creation of the generated file. See the **−i** keyletter for the *list* format.

−e Indicates that the *get* is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of *delta*(1). The **−e** keyletter used in a *get* for a particular version (SID) of the SCCS file prevents further *gets* for editing on the same SID until *delta* is executed or the **j** (joint edit) flag is set in the SCCS file [see *admin*(1)]. Concurrent use of **get −e** for different SIDs is always allowed.

If the *g-file* generated by *get* with an **−e** keyletter is accidentally ruined in the process of editing it, it may be regenerated by re-executing the *get* command with the **−k** keyletter in place of the **−e** keyletter.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file [see *admin*(1)] are enforced when the **−e** keyletter is used.

−b Used with the **−e** keyletter to indicate that the new delta should have an SID in a new branch as shown in Table 1. This keyletter is ignored if the **b** flag is not present in the file [see *admin*(1)] or if the retrieved *delta* is not a leaf *delta*. (A leaf *delta* is one that has no successors on the SCCS file tree.) Note: A branch *delta* may always be created from a non-leaf *delta*. Partial SIDs are interpreted as shown in the "SID Retrieved" column of Table 1.

−k Suppresses replacement of identification keywords (see below) in the retrieved text by their value. The **−k** keyletter is implied by the **−e** keyletter.

−l[p] Causes a delta summary to be written into an *l-file*. If **−lp** is used then an *l-file* is not created; the delta summary is written on the standard output instead. See *FILES* for the format of the *l-file*.

−p Causes the text retrieved from the SCCS file to be written on the standard output. No *g-file* is created. All output which normally goes to the standard output goes to file descriptor 2 instead, unless the **−s** keyletter is used, in which case it disappears.

−s Suppresses all output normally written on the standard output. However, fatal error messages (which always go to file descriptor 2) remain unaffected.

−m Causes each text line retrieved from the SCCS file to be preceded by the SID of the delta that inserted the text line in the SCCS file. The format is: SID, followed by a horizontal tab, followed by the text line.

−n Causes each generated text line to be preceded with the %M% identification keyword value (see below). The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the **−m** and **−n** keyletters are used, the format is: %M% value, followed by a horizontal tab, followed by the **−m** keyletter generated format.

−g Suppresses the actual retrieval of text from the SCCS file. It is primarily used to generate an *l-file*, or to verify the existence of a particular SID.

−t Used to access the most recently created delta in a given release (e.g., **−r1**), or release and level (e.g., **−r1.2**).

−w *string* Substitute *string* for all occurrences of %W% when getting the file.

−a*seq-no.* The delta sequence number of the SCCS file delta (version) to be retrieved [see *sccsfile*(5)]. This keyletter is used by the *comb*(1) command; it is not a generally useful keyletter. If both the **−r** and **−a** keyletters are specified, only the **−a** keyletter is used. Care should be taken when using the **−a** keyletter in conjunction with the **−e** keyletter, as the SID of the delta to be created may not be what one expects. The **−r** keyletter can be used with the **−a** and **−e** keyletters to control the naming of the SID of the delta to be created.

For each file processed, *get* responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the **−e** keyletter is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each file name is printed (preceded by a new-line) before it is processed. If the **−i** keyletter is used included deltas are listed following the notation "Included"; if the **−x** keyletter is used, excluded deltas are listed following the notation "Excluded".

TABLE 1. Determination of SCCS Identification String

SID* Specified	−b Keyletter Used†	Other Conditions	SID Retrieved	SID of Delta to be Created
none‡	no	R defaults to mR	mR.mL	mR.(mL + 1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB + 1).1
R	no	R > mR	mR.mL	R.1***
R	no	R = mR	mR.mL	mR.(mL + 1)
R	yes	R > mR	mR.mL	mR.mL.(mB + 1).1
R	yes	R = mR	mR.mL	mR.mL.(mB + 1).1
R	–	R < mR and R does *not* exist	hR.mL**	hR.mL.(mB + 1).1
R	–	Trunk succ.# in release > R and R exists	R.mL	R.mL.(mB + 1).1
R.L	no	No trunk succ.	R.L	R.(L + 1)
R.L	yes	No trunk succ.	R.L	R.L.(mB + 1).1
R.L	–	Trunk succ. in release ≥ R	R.L	R.L.(mB + 1).1
R.L.B	no	No branch succ.	R.L.B.mS	R.L.B.(mS + 1)
R.L.B	yes	No branch succ.	R.L.B.mS	R.L.(mB + 1).1
R.L.B.S	no	No branch succ.	R.L.B.S	R.L.B.(S + 1)
R.L.B.S	yes	No branch succ.	R.L.B.S	R.L.(mB + 1).1
R.L.B.S	–	Branch succ.	R.L.B.S	R.L.(mB + 1).1

* "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB+1).1" means "the first sequence number on the *new* branch (i.e., maximum branch number plus one) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components *must* exist.

** "hR" is the highest *existing* release that is lower than the specified, *nonexistent*, release R.

*** This is used to force creation of the *first* delta in a *new* release.

Successor.

† The **−b** keyletter is effective only if the **b** flag [see *admin*(1)] is present in the file. An entry of − means "irrelevant".

‡ This case applies if the **d** (default SID) flag is *not* present in the file. If the **d** flag *is* present in the file, then the SID obtained from the **d** flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

IDENTIFICATION KEYWORDS

Identifying information is inserted into the text retrieved from the SCCS file by replacing *identification keywords* with their value wherever they occur. The following keywords may be used in the text stored in an SCCS file:

Keyword	*Value*
%M%	Module name: either the value of the **m** flag in the file [see *admin*(1)], or if absent, the name of the SCCS file with the leading **s.** removed.
%I%	SCCS identification (SID) (%R%.%L%.%B%.%S%) of the retrieved text.
%R%	Release.
%L%	Level.
%B%	Branch.
%S%	Sequence.
%D%	Current date (YY/MM/DD).
%H%	Current date (MM/DD/YY).
%T%	Current time (HH:MM:SS).
%E%	Date newest applied delta was created (YY/MM/DD).
%G%	Date newest applied delta was created (MM/DD/YY).
%U%	Time newest applied delta was created (HH:MM:SS).

%Y% Module type: value of the **t** flag in the SCCS file [see *admin* (1)].

%F% SCCS file name.

%P% Fully qualified SCCS file name.

%Q% The value of the **q** flag in the file [see *admin* (1)].

%C% Current line number. This keyword is intended for identifying messages output by the program such as "this should not have happened" type errors. It is *not* intended to be used on every line to provide sequence numbers.

%Z% The 4-character string **@(#)** recognizable by *what* (1).

%W% A shorthand notation for constructing *what* (1) strings for UNIX system program files. %W% = %Z%%M%<horizontal-tab>%I%

%A% Another shorthand notation for constructing *what* (1) strings for non-UNIX system program files. %A% = %Z%%Y% %M% %I%%Z%

Several auxiliary files may be created by *get*. These files are known generically as the *g-file*, *l-file*, *p-file*, and *z-file*. The letter before the hyphen is called the tag. An auxiliary file name is formed from the SCCS file name: the last component of all SCCS file names must be of the form **s.***module-name*, the auxiliary files are named by replacing the leading **s** with the tag. The *g-file* is an exception to this scheme: the *g-file* is named by removing the **s.** prefix. For example, **s.xyz.c**, the auxiliary file names would be **xyz.c**, **l.xyz.c**, **p.xyz.c**, and **z.xyz.c**, respectively.

The *g-file*, which contains the generated text, is created in the current directory (unless the **−p** keyletter is used). A *g-file* is created in all cases, whether or not any lines of text were generated by the *get*. If the **−k** keyletter is used or implied its mode is 644; otherwise its mode is 444. Only the real user need have write permission in the current directory.

The *l-file* contains a table showing which deltas were applied in generating the retrieved text. The *l-file* is created in the current directory if the **−l** keyletter is used; its mode is 444 and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the *l-file* have the following format:

a. A blank character if the delta was applied;
   * otherwise.
b. A blank character if the delta was applied or was not applied and ignored;
   * if the delta was not applied and was not ignored.
c. A code indicating a "special" reason why the delta was or was not applied:
   "I": Included.
   "X": Excluded.
   "C": Cut off (by a **−c** keyletter).
d. Blank.
e. SCCS identification (SID).
f. Tab character.
g. Date and time (in the form YY/MM/DD HH:MM:SS) of creation.
h. Blank.
i. Login name of person who created *delta*.

The comments and **MR** data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

The *p-file* is used to pass information resulting from a *get* with an **−e** keyletter along to *delta*. Its contents are also used to prevent a subsequent execution of *get* with an **−e** keyletter for the same SID until *delta* is executed or the joint edit flag, **j**, [see *admin*(1)] is set in the SCCS file. The *p-file* is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. The format of the *p-file* is: the gotten SID, followed by a blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time the *get* was executed, followed by a blank and the **−i** keyletter argument if it was present, followed by a blank and the **−x** keyletter argument if it was present, followed by a new-line. There can be an arbitrary number of lines in the *p-file* at any time; no two lines can have the same new delta SID.

The *z-file* serves as a *lock-out* mechanism against simultaneous updates. Its contents are the binary (2 bytes) process ID of the command (i.e., *get*) that created it. The *z-file* is created in the directory containing the SCCS file for the duration of *get*. The same protection restrictions as those for the *p-file* apply for the *z-file*. The *z-file* is created mode 444.

FILES

g-file	Existed before the execution of *delta*; removed after completion of *delta*.
p-file	Existed before the execution of *delta*; may exist after completion of *delta*.
q-file	Created during the execution of *delta*; removed after completion of *delta*.
x-file	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z-file	Created during the execution of *delta*; removed during the execution of *delta*.
d-file	Created during the execution of *delta*; removed after completion of *delta*.
/usr/bin/bdiff	Program to compute differences between the "gotten" file and the *g-file*.

SEE ALSO

admin(1), delta(1), help(1), prs(1), what(1).

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

If the effective user has write permission (either explicitly or implicitly) in the directory containing the SCCS files, but the real user does not, then only one file may be named when the **−e** keyletter is used.

This page is intentionally left blank

NAME

help – SCCS Utility Help Facility

SYNOPSIS

help arg ...

DESCRIPTION

The Source Code Control System (SCCS) *help* provides assistance for use of SCCS commands and bdiff.

An argument can be a SCCS command name or an error code returned from one of the SCCS programs.

If the argument is a SCCS command name (e.g. get) or bdiff then *help* shows the synopsis for the command.

If the argument is an error code then *help* shows some explanation of this error code. The error codes consists of two letters followed by a number (e.g. ge3).

SEE ALSO

admin(1), bdiff(1), cdc(1), comb(1), delta(1), get(1), prs(1), rmdel(1), sact(1), sccsdiff(1), unget(1), val(1), vc(1), what(1), sccsfile(4).

This page is intentionally left blank

NAME

ld – link editor for common object files
uld – ucode link editor

SYNOPSIS

ld [option] . . . file . . .

uld [option] . . . file . . .

DESCRIPTION

The *ld* command combines several object files into one, performs relocation, resolves external symbols, and supports symbol table information for symbolic debugging. In the simplest case, the names of several object *files* are given. *ld* combines them, producing an object module that can be executed or used as input for a subsequent *ld* run. (In the latter case, the **–r** option must be given to preserve the relocation entries.) The output of *ld* is left in **a.out**. By default, this file is executable if no errors occurred during the load.

The argument object files are concatenated in the order specified. The entry point of the output is the beginning of the text segment (unless the **–e** option is specified).

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. The library (archive) symbol table (see *ar*(4)) is searched to resolve external references that can be satisfied by library members. Thus, the ordering of library members is unimportant.

When searching for libraries the default directories searched are **/lib**, **/usr/lib/cmplrs/cc** and **/usr/local/lib**.

The *uld* command combines several ucode object files and libraries into one ucode object file. It "hides" external symbols for better optimizations by subsequent compiler passes. The symbol tables of *coff* object files loaded with ucode object files are used to determine what external symbols not to "hide" along with files specified by the user that contain lists of symbol names.

All options are recognized by both *ld* and *uld* . Those options used by one and not the other are ignored. Any option can be preceded by a '**k**' (for example **−ko** outfile) and except for **−kl***x* have the same meaning with or without the preceding '**k**'. This is done so that these options can be passed to both link editors through compiler drivers.

The symbols 'etext', 'edata', 'end', '_ftext', '_fdata', '_fbss', '_gp', '_procedure_table', '_procedure_table_size' and '_procedure_string_table' are reserved. These loader defined symbols if referred to, are set their values as described in *end*(3). It is erroneous to define these symbols.

−e *epsym* Set the default entry point address for the output file to be that of the symbol *epsym*.

−l*x* Search a library **lib***x***.a**, where *x* is up to seven characters. A library is searched when its name is encountered, so the placement of a **−l** is significant.

−kl*x* Search a library *lib x .b*, where *x* is a string. These libraries are intended to be ucode object libraries. In all other ways, this option is like the −*l*x option.

−m Produce a map or listing of the input/output sections on the standard output.

−o *outfile* Produce an output object file by the name *outfile*. The name of the default object file is **a.out**.

−r Retain relocation entries in the output object file. Relocation entries must be saved if the output file is to become an input file in a subsequent *ld* run. Unless **−a** is also given, the link editor does not complain about unresolved references.

−s Strip the symbol table information from the output object file.

−u *symname* Enter *symname* as an undefined symbol in the symbol table.

This is useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

−L *dir* Change the algorithm of searching for **lib***x***.a** to look in *dir* before looking in **/lib** and **/usr/lib**.

This option is effective only if it precedes the **−l** option on the command line.

−L Change the algorithm of searching for lib*x*.a or libx.b to never look in the default directories.

This is useful when the default directories for libraries should not be searched and only the directories specified by **−L***dir* are to be searched.

−N Put the data section immediately following the text in the output file.

−V Output a message giving information about the version of *ld* being used.

−VS *num* Use *num* as a decimal version stamp identifying the *a.out* file that is produced. The version stamp is stored in the optional header.

−K*dir* Change the default directories to the single directory *dir*. This option is only intended to be used by the compiler driver. Users should use the **−L** and **−L**dir options to get the effect they desire.

−B*string*
: Append *string* to the library names created for the **−l***x* and **−kl***x* when searching for library names. For each directory to be searched the name is first created with the *string* and if it is not found it is created without the *string*.

−p file
: Preserve (don't "hide") the symbol names listed in *file* when loading ucode object files. The symbol names in the file are separated by blanks, tabs, or newlines.

−x
: Do not preserve local (non−.globl) symbols in the output symbol table; enter external and static symbols only. This option saves some space in the output file.

−d
: Force definition of common storage and define loader defined symbols even if −r is present.

−F or **−z**
: Arrange for the process to be loaded on demand from the resulting executable file (413 format) rather than preloaded, a ZMAGIC file. This is the default.

−n
: Arrange (by giving the output file a 0410 "magic number") that when the output file is executed, the text portion will be read-only and shared among all users executing the file, an NMAGIC file.

  The default text segment address is `0x00400000` and the default data segment address is `0x10000000`.

−nM
: arrange (by giving the output file a `0410` "magic number") that when the output file is executed, the text portion will be read-only and shared among all users executing the file, an NMAGIC file. This involves moving the data areas up to the first possible

pagesize byte boundary following the end of the text.

−N Place the data section immediately after the text and do not make the text portion read only or sharable, an OMAGIC file. (Use "magic number" 0407.)

−T num Set the text segment origin. The argument *num* is a hexadecimal number.

−D num Set the data segment origin. The argument *num* is a hexadecimal number. See the NOTES section for restrictions.

−B num Set the bss segment origin. The argument *num* is a hexadecimal number. This option can be used only if the final object is an OMAGIC file.

−S Set silent mode and suppress non-fatal errors.

−v Set verbose mode. Print the name of each file as it is processed.

−ysym Indicate each file in which *sym* appears, *sym*'s type and whether the file defines or references *sym*. Many such options may be given to trace many symbols.

−f fill Set the fill pattern for "holes" within an output section. The argument *fill* is a four-byte hexadecimal constant.

−G num The argument *num* is taken to be a decimal number that is the largest size in bytes of a *.comm* item or literal that is to be allocated in the small bss section for reference off the global pointer. The default is 8 bytes.

−bestGnum Calculate the best **−G** *num* to use when compiling and linking the files which produced the objects being linked. Using too large a number with the **−G** *num* option may cause the gp (global-pointer) data area to overflow; using too small a number may reduce your program's execution speed.

−count, **−nocount**, **−countall**
These options control which objects are counted as recompilable for the best **−G** *num* calculation. By default, the **−bestGnum** option assumes you can recompile everything with a different **−G** *num* option. If you cannot recompile certain object files or libraries (because, for example, you have no sources for them), use these options to tell the link editor to take this into account in calculating the best **−G** *num* value. **−nocount** says that object files appearing after it on the command line cannot be recompiled; **−count** says that object files appearing after it on the command line can be recompiled; you can alternate the use of **−nocount** and **−count.** **−countall** overrides any **−nocount** options appearing after it on the command line.

−b Do not merge the symbolic information entries for the same file into one entry for that file. This is only needed when the symbolic information from the same file appears differently in any of the objects to be linked. This can occur when object files are compiled, by means of conditional compilation, with an apparently different version of an include file.

−jmpopt and **−nojmpopt**
Fill or don't fill the delay slots of jump instructions with the target of the jump and adjust the jump offset to jump past that instruction. This always is disabled for debugging (when the **−g1, −g2** or **−g** flag is present). When this option is enabled it requires that all of the loaded program's text be in memory and could cause the loader to run out of memory. The default is **−nojmpopt.**

−g or **−g[0123]** These options are accepted and except for **−g1, −g2** or **−g** disabling the **−jmpopt** have no other effect.

−A *file* This option specifies incremental loading, i.e. linking is to be done in a manner so that the resulting object may be read into an already executing program. The next argument, *file,* is the name of a file whose symbol table will be taken as a basis on which to define additional symbols. Only newly linked material will be entered into the text and data portions of **a.out,** but the new symbol table will reflect every symbol defined before and after the incremental load. This argument must appear before any other object file in the argument list. The **−T** option may be used as well, and will be taken to mean that the newly linked segment will commence at the corresponding address (which must be a correct multiple for the resulting object type). The default resulting object type is an OMAGIC file and the default starting address of the text is the old value of end rounded to SCNROUND as defined in the include file *<scnhdr.h>.* Using the defaults, when this file is read into an already executing program the initial value of the break must also be rounded. All other objects except the argument to the **−A** option must be compiled **−G** 0 and this sets **−G** 0 for linking.

−EL and **−EB** Are ignored. Big Endian is default.

FILES

/bin/ld The linker driver.

$COMP_HOST_ROOT/usr/lib/cmplrs/ld
The linker for TARGETMC R3KMI.

/lib/lib*.a,
/usr/lib*.a
/usr/local/lib/lib*.a Libraries.

a.out output file

SEE ALSO

as(1), cc(1), a.out(4), ar(4).

NOTES

The segments must not overlap.

All addresses must be less than `0x80000000`. The stack starts below `0x80000000` and grows through lower addresses so space should be left for it.

For ZMAGIC and NMAGIC files the default text segment address is `0x00400000` and the default data segment is `0x10000000`. For OMAGIC files the default text segment address is `0x10000000` with the data segment following the text segment.

The default for all types of files is that the bss segment follows the data segment.

For OMAGIC files to be run under the operating system the **−B** flag should not be used because the bss segment must follow the data segment which is the default.

For OMAGIC files, the −B flag should not be used because the bss segment must follow the data segment which is default.

WARNINGS

Through its options and input directives, the common link editor gives users great flexibility; however, those who use the input directives must assume some added responsibilities. Input directives should insure the following properties for programs:

- C defines a zero pointer as null. A pointer to which zero has been assigned must not point to any object. To satisfy this, users must not place any object at virtual address zero in the data space.
- When the link editor is called through *cc*(1), a startup routine is linked with the user's program. This routine calls exit ( ) (see *exit*(2)) after execution of the main program. If the user calls the link editor directly, then the user must insure that the program always calls exit( ) rather than falling through the end of the entry routine.

This page is intentionally left blank

NAME

lex – generate programs for simple lexical tasks

SYNOPSIS

lex [**–rctvn**] [file] ...

DESCRIPTION

The *lex* command generates programs to be used in simple lexical analysis of text.

The input *files* (standard input default) contain strings and expressions to be searched for, and C text to be executed when strings are found.

A file **lex.yy.c** is generated which, when loaded with the library, copies the input to the output except when a string specified in the file is found; then the corresponding program text is executed. The actual string matched is left in *yytext*, an external character array. Matching is done in order of the strings in the file. The strings may contain square brackets to indicate character classes, as in **[abx–z]** to indicate **a**, **b**, **x**, **y**, and **z**; and the operators ∗, +, and **?** mean respectively any non-negative number of, any positive number of, and either zero or one occurrence of, the previous character or character class. The character **.** is the class of all ASCII characters except new-line. Parentheses for grouping and vertical bar for alternation are also supported. The notation *r*{*d*,*e*} in a rule indicates between *d* and *e* instances of regular expression *r*. It has higher precedence than |, but lower than ∗, ?, +, and concatenation. Thus **[a–zA–Z]+** matches a string of letters. The character ˆ at the beginning of an expression permits a successful match only immediately after a new-line, and the character **$** at the end of an expression requires a trailing new-line. The character **/** in an expression indicates trailing context; only the part of the expression up to the slash is returned in *yytext*, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within " symbols or preceded by \.

Three subroutines defined as macros are expected: **input()** to read a character; **unput**(*c*) to replace a character read; and **output**(*c*) to place an output character. They are defined in terms of the standard streams, but you can override them. The program generated is named **yylex()**, and the library contains a **main()** which calls it. The action REJECT on the right side of the rule causes this match to be rejected and the next suitable match executed; the function **yymore()** accumulates additional characters into the same *yytext*; and the function **yyless**(*p*) pushes back the portion of the string matched beginning at *p*, which should be between *yytext* and *yytext* + *yyleng*. The macros *input* and *output* use files **yyin** and **yyout** to read from and write to, defaulted to **stdin** and **stdout**, respectively.

Any line beginning with a blank is assumed to contain only C text and is copied; if it precedes %% it is copied into the external definition area of the **lex.yy.c** file. All rules should follow a %%, as in YACC. Lines preceding %% which begin with a non-blank character define the string on the left to be the remainder of the line; it can be called out later by surrounding it with {}. Note that curly brackets do not imply parentheses; only string substitution is done.

EXAMPLE

```
D       [0-9]
%%
if      printf("IF statement\n");
[a-z]+  printf("tag, value %s\n",yytext);
0{D}+   printf("octal number %s\n",yytext);
{D}+    printf("decimal number %s\n",yytext);
"++"    printf("unary op\n");
"+"     printf("binary op\n");
"/*"      skipcommnts();
%%
 skipcommnts()
 {
        for (;;)
        {
                while (input() != '*')
```

```
                    ;
                if (input() != '/')
                        unput(yytext[yyleng-1]);
                else
                        return;
        }
}
```

The external names generated by *lex* all begin with the prefix **yy** or **YY**.

The flags must appear before any files. The flag **−r** indicates RATFOR actions, **−c** indicates C actions and is the default, **−t** causes the **lex.yy.c** program to be written instead to standard output, **−v** provides a one-line summary of statistics, **−n** will not print out the **−v** summary. Multiple files are treated as a single file. If no files are specified, standard input is used.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

%**p** *n*	number of positions is *n* (default 2500)
%**n** *n*	number of states is *n* (500)
%**e** *n*	number of parse tree nodes is *n* (1000)
%**a** *n*	number of transitions is *n* (2000)
%**k** *n*	number of packed character classes is *n* (1000)
%**o** *n*	size of output array is *n* (3000)

The use of one or more of the above automatically implies the **−v** option, unless the **−n** option is used.

SEE ALSO

yacc(1).
Programmer's Guide.

BUGS

The **−r** option is not yet fully operational.

This page is intentionally left blank

NAME

lint – a C program checker

SYNOPSIS

lint [option] ... file ...

DESCRIPTION

The *lint* command attempts to detect features of the C program files that are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compilers. Among the things that are currently detected are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions that return values in some places and not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but none returned.

Arguments whose names end with **.c** are taken to be C source files. Arguments whose names end with **.ln** are taken to be the result of an earlier invocation of *lint* with either the **–c** or the **–o** option used. The **.ln** files are analogous to **.o** (object) files that are produced by the *cc*(1) command when given a **.c** file as input. Files with other suffixes are warned about and ignored.

lint will take all the **.c**, **.ln**, and **llib-l***x***.ln** (specified by **–l***x*) files and process them in their command line order. By default, *lint* appends the standard C lint library (**llib-lc.ln**) to the end of the list of files. However, if the **–p** option is used, the portable C lint library (**llib-port.ln**) is appended instead. When the **–c** option is not used, the second pass of *lint* checks this list of files for mutual compatibility. When the **–c** option is used, the **.ln** and the **llib-l***x***.ln** files are ignored.

Any number of *lint* options may be used, in any order, intermixed with file-name arguments. The following options are used to suppress certain kinds of complaints:

−a Suppress complaints about assignments of long values to variables that are not long.

−b Suppress complaints about **break** statements that cannot be reached. (Programs produced by *lex* or *yacc* will often result in many such complaints).

−h Do not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

−u Suppress complaints about functions and external variables used and not defined, or defined and not used. (This option is suitable for running *lint* on a subset of files of a larger program).

−v Suppress complaints about unused arguments in functions.

−x Do not report variables referred to by external declarations but never used.

The following arguments alter *lint*'s behavior:

−l*x* Include additional lint library **llib-l***x***.ln**. For example, you can include a lint version of the math library **llib-lm.ln** by inserting **−lm** on the command line. This argument does not suppress the default use of **llib-lc.ln**. These lint libraries must be in the assumed directory. This option can be used to reference local lint libraries and is useful in the development of multi-file projects.

−n Do not check compatibility against either the standard or the portable lint library.

−p Attempt to check portability to other dialects (IBM and GCOS) of C. Along with stricter checking, this option causes all non-external names to be truncated to eight characters and all external names to be truncated to six characters and one case.

−c Cause *lint* to produce a **.ln** file for every **.c** file on the command line. These **.ln** files are the product of *lint*'s first pass only, and are not checked for inter-function compatibility.

−o lib Cause *lint* to create a lint library with the name **llib-l***lib***.ln**. The **−c** option nullifies any use of the **−o** option. The lint library produced is the input that is given to *lint*'s second pass. The **−o** option simply causes this file to be saved in the named lint library. To produce a **llib-l***lib***.ln** without extraneous messages, use of the **−x** option is suggested. The **−v** option is useful if the source file(s) for the lint library are just external interfaces (for example, the way the file **llib-lc** is written). These option settings are also available through the use of "lint comments" (see below).

The **−D**, **−U**, and **−I** options of *cpp*(1) and the **−g** and **−O** options of *cc*(1) are also recognized as separate arguments. The **−g** and **−O** options are ignored, but, by recognizing these options, *lint*'s behavior is closer to that of the *cc*(1) command.

Other options are warned about and ignored. The preprocessor symbol "lint" is defined to allow certain questionable code to be altered or removed for *lint*. Therefore, the symbol "lint" should be thought of as a reserved word for all code that is planned to be checked by *lint*.

Certain conventional comments in the C source will change the behavior of *lint*:

/ * NOTREACHED * / at appropriate points stops comments about unreachable code. [This comment is typically placed just after calls to functions like *exit*(2)].

/ * VARARGS*n* * / suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/ * ARGSUSED * / turns on the **−v** option for the next function.

/ * LINTLIBRARY * / at the beginning of a file shuts off complaints about unused functions and function arguments in this file. This is equivalent to using the **−v** and **−x** options.

lint produces its first output on a per-source-file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, if the **−c** option is not used, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source file name will be printed followed by a question mark.

The behavior of the **−c** and the **−o** options allows for incremental use of *lint* on a set of C source files. Generally, one invokes *lint* once for each source file with the **−c** option. Each of these invocations produces a **.ln** file which corresponds to the **.c** file, and prints all messages that are about just that source file. After all the source files have been separately run through *lint*, it is invoked once more (without the **−c** option), listing all the **.ln** files with the needed **−l***x* options. This will print all the inter-file inconsistencies. This scheme works well with *make*(1); it allows *make* to be used to *lint* only the source files that have been modified since the last time the set of source files were *lint'ed.*

FILES

LLIBDIR	the directory where the lint libraries specified by the **−l***x* option must exist, usually /usr/lib
LLIBDIR/lint[12]	first and second passes
LLIBDIR/llib-lc.ln	declarations for C Library functions (binary format; source is in *LLIBDIR*/llib-lc)
LLIBDIR/llib-port.ln	declarations for portable functions (binary format; source is in *LLIBDIR*/llib-port)
LLIBDIR/llib-lm.ln	declarations for Math Library functions (binary format; source is in *LLIBDIR*/llib-lm)
TMPDIR/ * lint *	temporaries
TMPDIR	usually /usr/tmp but can be redefined by setting the environment variable **TMPDIR** [see *tempnam*() in *tmpnam*(3S)].

SEE ALSO

cc(1), cpp(1), make(1).

BUGS

exit(2), *setjmp*(3C), and other functions that do not return are not understood; this causes various lies.

This page is intentionally left blank

NAME

list – produce C source listing from a common object file

SYNOPSIS

list [–V] [–**h**] [–**F** function] source-file **. . .**
[object-file]

DESCRIPTION

The *list* command produces a C source listing with line number information attached. If multiple C source files were used to create the object file, *list* will accept multiple file names. The object file is taken to be the last non-C source file argument. If no object file is specified, the default object file, **a.out**, will be used.

Line numbers will be printed for each line marked as breakpoint inserted by the compiler (generally, each executable C statement that begins a new line of source). Line numbering begins anew for each function. Line number 1 is always the line containing the left curly brace ({) that begins the function body. Line numbers will also be supplied for inner block redeclarations of local variables so that they can be distinguished by the symbolic debugger.

The following options are interpreted by *list* and may be given in any order:

–V Print, on standard error, the version number of the *list* command executing.

–h Suppress heading output.

–F*function* List only the named function. The **–F** option may be specified multiple times on the command line.

SEE ALSO

as(1), cc(1), ld(1).

CAVEATS

Object files given to *list* must have been compiled with the **–g** option of *cc*(1).

Since *list* does not use the C preprocessor, it may be unable to recognize function definitions whose syntax has been distorted by the use of C preprocessor macro substitutions.

DIAGNOSTICS

list will produce the error message "list: name: cannot open" if *name* cannot be read. If the source file names do not end in **.c** , the message is "list: name: invalid C source name". An invalid object file will cause the message "list: name: bad magic" to be produced. If some or all of the symbolic debugging information is missing, one of the following messages will be printed: "list: name: symbols have been stripped, cannot proceed", "list: name: cannot read line numbers", and "list: name: not in symbol table". The following messages are produced when *list* has become confused by **#ifdef's** in the source file: "list: name: cannot find function in symbol table", "list: name: out of sync: too many }", and "list: name: unexpected end-of-file". The error message "list: name: missing or inappropriate line numbers" means that either symbol debugging information is missing, or *list* has been confused by C preprocessor statements.

NAME

lorder – find ordering relation for an object library

SYNOPSIS

lorder file ...

DESCRIPTION

The input is one or more object or library archive *files* (see *ar*(1)). The standard output is a list of pairs of object file or archive member names, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by *tsort*(1) to find an ordering of a library suitable for one-pass access by *ld*(1). Note that the link editor *ld*(1) is capable of multiple passes over an archive in the portable archive format (see *ar*(4)) and does not require that *lorder*(1) be used when building an archive. The usage of the *lorder*(1) command may, however, allow for a slightly more efficient access of the archive during the link edit process.

If more than one filename are specified the files must be of the same code-type, i.e code generated with the same values of the TARGETMC environment – see *intro*(1).

The following example builds a new library from existing **.o** files.

```
ar −cr library lorder  *.o | tsort
```

FILES

TMPDIR/ * symref	temporary files
TMPDIR/ * symdef	temporary files

TMPDIR is usually /usr/tmp but can be redefined by setting the environment variable **TMPDIR** (see *tempnam*() in *tmpnam*(3S)).

SEE ALSO

ar(1), ld(1), tsort(1), ar(4).

CAVEAT

lorder will accept as input any object or archive file, regardless of its suffix, provided there is more than one input file. If there is but a single input file, its suffix must be **.o**.