Data General

Customer Documentation

Programmer's Reference for the DG/UX™ System (Volume 1)

AViiON®

PRODUCT LINE

Programmer's Reference for the DG/UX™ System (Volume 1)

093-701055-02

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-701055

Printed in the United States of America
Revision 02, June 1991

Programmer's Reference for the DG/UX System (Volume 1)

093-701055-02

Revision History:	Effective with:
Original Release – February 1990	DG/UX 4.20
Revision 1 – June 1990	DG/UX 4.30
Revision 2 – June 1991	DG/UX 5.4

Preface

This is Volume 1 of the *Programmer's Reference for the DG/UX™ System*. The *Programmer's Reference* describes the programming features of the DG/UX system. It contains individual manual pages that describe commands, system calls, subroutines, file formats, and other useful topics, such as the ASCII table shown on **ascii**(5).

This manual is part of a five-volume reference set. The other manuals are the *System Manager's Reference for the DG/UX System* and the *User's Reference for the DG/UX System*. These manuals contain in printed (typeset) form the online entries released with the DG/UX System in **/usr/catman** for access by the **man** command.

The *Programmer's Reference* provides neither a general overview of the DG/UX system nor details of the implementation of the system. For more details about some of the most often used programming tools, see *Programmer's Guide: ANSI C and Programming Support Tools*, *Programmer's Guide: System Services and Application Packaging Tools*, and the Data General supplements to these two manuals. Other related manuals are listed under "Related Manuals" at the end of this manual.

Man Pages

For historical reasons, each entry is called a "manual page" or "man page," though an entry may occupy more than one physical page and may contain more than one entry. If the man page contains more than one entry, it is alphabetized under its "primary" name; for example, the **uname** manual page describes the **uname** and **nuname** files.

Manual pages are assigned to classes ranging from 0 through 8 for easy cross-reference. The class number appears in parentheses following the name; for example, in **accept**(1M) the "1" indicates that **accept** is a command, and the "M" indicates that the man page is in the *System Manager's Reference*.

A command followed by a (1) or (1G) usually means that it is described in the *User's Reference*. (Class 1 commands appropriate for use by programmers are located in the *Programmer's Reference*.) A man page name with a (1M), (4M), (7), or (8) following it means that the entry is in the *System Manager's Reference*. Names with (2) or (3*x*), (4), (5) [except **editread**(5)], or (6F) are in the *Programmer's Reference*. Occasionally, DG/UX man pages refer to other products' man pages, which are not part of the DG/UX documentation; these are so noted.

Manual Organization

Volume 1 contains two chapters:

Chapter 1: Commands (1)
This chapter describes commands that support C and other programming languages.

Chapter 2: System Calls (2) This chapter describes the access to services provided by the DG/UX kernel, including the C language interface and a description of returned error codes.

Volume 2 contains one chapter:

Chapter 3: Subroutines and Libraries (3) This chapter describes the available subroutines and subroutine libraries. Their binary versions reside in various system libraries in the directories **/lib** and **/usr/lib**. See **intro**(3) for descriptions of these libraries and the files in which they are stored. Although these man pages are alphabetized together, each has a letter associated with the number 3 indicating the pertinent library:

- 3C C Programming Language Libraries
- 3E ELF Library Routines
- 3G General Library Routines
- 3M Mathematical Library Routines
- 3N Networking Support Utilities
- 3S Standard I/O Library Routines
- 3X Specialized Libraries

Volume 3 contains three chapters and one appendix:

Chapter 4: File Formats (4) This chapter documents the structure of particular kinds of files; for example, the format of the output of the link editor is given in **a.out**(4). Excluded are files used by only one command (for example, the assembler's intermediate files). In general, the C language structures corresponding to these formats can be found in the directories **/usr/include** and **/usr/include/sys**.

Chapter 5: Miscellaneous Features (5) This chapter contains a variety of facilities. Included are descriptions of character sets, macro packages, and other things.

Chapter 6: Communications Protocols (6) This chapter contains a description of the **unix_ipc** communications facility.

Appendix A: Contents and Permuted Index Man Pages
These manual pages contain information extracted from the DG/UX man pages in all five reference volumes.

Man Page Format

Each man page has at least some of the following sections:

NAME	gives the primary name (and secondary names, as the case may be) and briefly states its purpose.
SYNOPSIS	summarizes the usage of the program being described.
DESCRIPTION	discusses how to use these commands.
EXAMPLES	gives examples of usage, where appropriate.
FILES	contains the file names that are referenced by the program.
EXIT CODES	discusses values set when the command terminates. The value set is available in the shell environment variable "?" (see **sh**(1)).
DIAGNOSTICS	discusses the error messages that may be produced. Messages that are intended to be self-explanatory are not listed.
SEE ALSO	offers pointers to related information.
NOTES	gives information that may be helpful under the particular circumstances described.

Some man pages may contain other heads such as **ENVIRONMENT** and **CAVEATS**.

Man Page Notation Conventions

This manual uses certain symbols and styles of type to indicate different meanings in man pages. Those symbol and typeface conventions are defined in the following list. You should familiarize yourself with these conventions before reading the manual.

The description of convention meanings uses the terms "command line," "format line," and "syntax line." A command line is an example of a command string that you should type verbatim; it is preceded by a system prompt. A format line shows how to structure a command; it shows the variables that must be supplied and the available options. A syntax line is a fragment of program code that shows how to use a particular routine; some syntax lines contain variables.

Convention	Meaning
boldface	This font is used for section heads and subsection heads. It is also used to distinguish input from output in examples where the two are intermixed.
`constant width/ monospace`	In command formats and code syntax: This typeface indicates text (including punctuation) that you type verbatim from your keyboard. In text: This typeface is used for examples, code samples, pathnames, and the names of commands, files, directories, and manual pages. In all contexts: The following characters, which have special meanings explained below, do not have special meaning but simply represent themselves when they appear in constant-width font: `< > [ ] { } |`. In constant-width font they are are I/O redirection operators, brackets, braces, and the pipe symbol.
italic	In format lines: This font represents variables for which you supply values; for example, the names of your directories and files, your username and password, and possible arguments to commands.
[*optional*]	In format lines: Regular-font brackets surround an optional argument. Don't type the brackets; they only set off what is optional. These brackets should not be confused with constant-width brackets.
choice1 | *choice2*	In format lines: The vertical bar indicates a choice between *choice1* and *choice2*.
...	In format lines and syntax lines: You can repeat the preceding argument as many times as desired.
{ }	In format lines: These regular-font braces surround either two or more choices or syntax elements that are repeatable as a group.
< >	In command lines and other examples: Angle brackets distinguish a command sequence or a keystroke (such as **<Ctrl-D>**, **<Esc>**, and **<3dw>**) from surrounding text. Note that these angle brackets are in regular type and that you do not type them; there are, however, constant-width versions of these symbols that you do type.
`$, %, #`	In command lines and other examples: These symbols represent the system command prompt symbols used for the Bourne and Korn shells, the C shell, and the superuser, respectively. Note that your system might use different symbols for the command prompts.

Contacting Data General

Data General wants to assist you in any way it can to help you use its products. Please feel free to contact the company as outlined below.

Manuals

If you require additional manuals, please use the enclosed TIPS order form (United States only) or contact your local Data General sales representative. A list of related documents appears at the end of this manual with the TIPS order form.

For a complete list of AViiON® and DG/UX™ manuals, see the *Guide to AViiON® and DG/UX™ System Documentation* (069-701085). The on-line version of this manual found in **/usr/release/doc_guide** contains the most current list.

Telephone Assistance

If you are unable to solve a problem using any manual you received with your system, free telephone assistance is available with your hardware warranty and with most Data General software service options. If you are within the United States or Canada, contact the Data General Service Center by calling 1-800-DG-HELPS. Lines are open from 8:00 a.m. to 5:00 p.m., your time, Monday through Friday. The center will put you in touch with a member of Data General's telephone assistance staff who can answer your questions.

For telephone assistance outside the United States or Canada, ask your Data General sales representative for the appropriate telephone number.

Joining Our Users Group

Please consider joining the largest independent organization of Data General users, the North American Data General Users Group (NADGUG). In addition to making valuable contacts, members receive FOCUS monthly magazine, a conference discount, access to the Software Library and Electronic Bulletin Board, an annual Member Directory, Regional and Special Interest Groups, and much more. For more information about membership in the North American Data General Users Group, call 1-800-877-4787 or 1-512-345-5316.

End of Preface

Contents

Chapter 1 — Commands

Chapter 2 — System Calls

Index

Related Documents

Chapter 1
Commands

This chapter contains in printed form all the online manual entries for programming-related DG/UX commands. Except for **intro**(1), the entries are in alphabetical order.

NAME

`intro` – introduction to commands and application programs

DESCRIPTION

This section describes, in alphabetical order, publicly-accessible commands.

Command Syntax

Unless otherwise noted, commands described in this section accept options and other arguments according to the following syntax:

name [*option*(*s*)] [*cmdarg*(*s*)]

name The name of an executable file.

option – *noargletter*(*s*) or,
– *argletter*<>*optarg*
where <> is optional white space.

noargletter A single letter representing an option without an argument.

argletter A single letter representing an option requiring an argument.

optarg Argument (character string) satisfying preceding *argletter*.

cmdarg Path name (or other command argument) *not* beginning with – or, – by itself indicating the standard input.

Command Syntax Standard: Rules

All new commands will follow the syntax rules below. Because existing commands have been developed at various times by various people, some commands will not follow the rules below. `Getopts`(1) should be used by all shell procedures to parse positional parameters and to check for legal options. `Getopts`(1) supports Rules 3-10 below. The command itself must enforce the other rules.

1. Command names (*name* above) must be between two and nine characters long.
2. Command names must include only lower-case letters and digits.
3. Option names (*option* above) must be one character long.
4. All options must be preceded by "–".
5. Options with no arguments may be grouped after a single "–".
6. The first option-argument (*optarg* above) following an option must be preceded by white space.
7. Option-arguments cannot be optional.
8. Groups of option-arguments following an option must either be separated by commas or separated by white space and quoted (e.g., `-o xxx,z,yy` or `-o "xxx z yy"`).
9. All options must precede operands (*cmdarg* above) on the command line.
10. "––" may be used to indicate the end of the options.
11. The order of the options relative to one another should not matter.
12. The relative order of the operands (*cmdarg* above) may affect their significance in ways determined by the command with which they appear.
13. "–" preceded and followed by white space should only be used to mean standard input.

DIAGNOSTICS

Upon termination, each command returns two bytes of status, one supplied by the system and giving the cause for termination, and (in the case of normal termination) one supplied by the program (see `wait`(2) and `exit`(2)). The former byte is 0 for normal termination; the latter is customarily 0 for successful execution and non-zero to indicate troubles such as erroneous parameters, bad or inaccessible data, or other inability to cope with the task at hand. It is called variously "exit code," "exit status," or "return code," and is described only where special conventions are involved.

SEE ALSO

`getopts`(1), `exit`(2), `wait`(2), `getopt`(3C).

NOTES

Many commands do not adhere to the aforementioned syntax.

Some commands produce unexpected results when processing files containing null characters. These commands often treat text input lines as strings and therefore become confused upon encountering a null character (the string terminator) within a line.

NAME

admin – create and administer SCCS files

SYNOPSIS

admin [-n] [-i[*name*]] [-r*rel*] [-t[*name*]] [-f*flag*[*flag-val*]] [-d*flag*[*flag-val*]] [-l*list*] [-a*login*] [-e*login*] [-m[*mrlist*]] [-y[*comment*]] [-h] [-z] *files*

DESCRIPTION

Admin creates new SCCS files and changes parameters of existing ones. 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 any options specified. Parameters not initialized are assigned a default value. If a named file does exist, parameters corresponding to specified options are changed, and other parameters are left as they are.

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 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 ignored.

The options 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 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 for delta numbering scheme). If the i option 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 option is omitted, then the SCCS file is created empty. Only one SCCS file may be created by an admin command line including the i option. Using a single admin to create two or more SCCS files requires that they be created empty (no -i option). Note that the -i option implies the -n option.

-r*rel* The release into which the initial delta is inserted. This option may be used only if the -i option is also used. If the -r option 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 option is used and admin is creating a new SCCS file (the -n and/or -i options also used), the descriptive text file name must also be supplied. In the case of existing SCCS files: (1) a -t option without a file name removes descriptive text (if any) currently in the SCCS file, and (2) a -t option with a file name substitutes text (if any) in the named file for the descriptive text (if any) currently in the SCCS file.

-f*flag* Specifies a flag, and, possibly, a value for the flag, to be placed in the SCCS file. Several f options may be supplied on a single admin command line. The allowable flags and their values are:

b Allows use of the -b option on a get(1) command to create branch deltas.

c*ceil* The highest release (ceiling), a number less than or equal to 9999, that can be retrieved by a `get(1)` command for editing. The default value for an unspecified `c` flag is 9999.

f*floor* The lowest release (floor), a number greater than 0 but less than 9999, that can 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*] Treats 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 must not contain 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 ::= *RELEASE NUMBER* | a

The character `a` in the list is equivalent to specifying "all releases" for the named SCCS file.

n Makes `delta(1)` create a "null" delta in any releases 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. If you don't use this flag, skipped releases won't show up in the SCCS file, thus 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* Module 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*] Makes `delta(1)` 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 you set this flag when creating an SCCS file, you must also use the `m` option, even if its value is null).

-d*flag* Removes (deletes) the specified *flag* from an SCCS file. You may specify this option only when processing existing SCCS files. Several `-d` options may be supplied on a single `admin` command. See the `-f` option for allowable *flag* names.

`-l`*list* A *list* of releases to be "unlocked." See the `-f` option for a description of the `l` flag and the syntax of a *list*.

`-a`*login* A *login* name, or numerical group ID, to be added to the list of users who may make deltas (changes) to the SCCS file. A group ID is equivalent to all *login* names common to that group ID. Several `a` options may be used on a single `admin` command line. As many *logins*, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas. To deny the privilege to a *login* or group ID, put a `!` in front of it; e.g., `-a!fred` will assert that `fred` may not add 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` options may be used on a single `admin` command line.

`-m`[*mrlist*]
The list of Modification Request (*MR*) numbers is inserted into the SCCS file as the reason for creating the initial delta, just as for `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, just as for `delta`(1). Omitting the `-y` option 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` option is valid only if the `-i` and/or `-n` options are specified (i.e., a new SCCS file is being created).

`-h` Makes `admin` check the structure of the SCCS file (see `sccsfile`(5)), and compare the sum of all the characters in the SCCS file, except those in the first line, with the check-sum stored in the first line of the SCCS file. Appropriate error diagnostics are produced.

This option inhibits writing on the file, so that it nullifies the effect of any other options supplied. It is meaningful only 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).

Using this option on a truly corrupted file may prevent future detection of the corruption.

EXAMPLES

```
admin -ifile1 s.file1
```

This command will take a file called 'file1' and create an SCCS file named 's.file1'. NOTE: If you receive a message 'No id keywords (cm7)' do not be alarmed, it is a warning message and should be ignored for now.

```
admin -ifile2 -r2.02 s.file2
```

This command will take a file called 'file2' and create an SCCS file named 's.file2', which will have a release of 2.02. Once again if you should receive message 'No id keywords (cm7)' do not be alarmed, it is just a warning message and should be ignored for now.

```
admin -ajohn s.file3
```

This command allows user 'john' to make deltas (changes) to the SCCS file 's.file3', while the command `admin -ejohn s.file3` revokes the privilege for john to change the file 's.file3'.

FILES

The last component of all SCCS path names must be of the form `s.`*filename*. New SCCS files are given mode 444 (see `chmod(1)`). Write permission in the pertinent directory is required to create a file. All writing done by `admin` is to a temporary x-file, called `x.`*filename*, (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.

Directories containing SCCS files should have access mode 755 and SCCS files themselves should be mode 444. This mode of the directories lets only the owner modify SCCS files in the directories. The mode of the SCCS files prevents any modification at all except by SCCS commands.

If you need to patch an SCCS file for any reason, the mode may be changed to 644 by the owner allowing use of `ed(1)`. *Be careful!* 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. Use another `admin -h` to ensure that the SCCS file is valid.

`Admin` also uses a transient lock file (called `z.`*filename*), which prevents simultaneous updates to the SCCS file by different users. See `get(1)` for more information.

DIAGNOSTICS

Use `help(1)` for explanations.

SEE ALSO

`delta(1)`, `ed(1)`, `get(1)`, `help(1)`, `prs(1)`, `what(1)`, `sccsfile(4)`.

NAME

ar – archive and library maintainer for portable archives

SYNOPSIS

ar [-V] [–]*key* [*posname*] *afile* [*name*] ...

where:

key One of the following letters: drqtpmx. Arguments to *key* are made with one of more of the following set: vuaibcls.

posname An archive member name used as a reference point in positioning other files in the archive.

afile The name of the archive file.

name A constituent file in the archive file.

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 are 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 ar(1) 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.

Options

–V Print ar's version number on standard error.

Key Characters

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, 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, 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 other key arguments are as follows:

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. When used with `x`, precede each file with a name.

u Act only on those files with dates of modification later than the archive file's.

a (See the `r` key letter.)

b (See the `r` key letter.)

i (See the `r` key letter.)

c Suppress the message that is produced by default when *afile* is created.

l Place temporary files in the local (current working) directory rather than in the default temporary directory, *TMPDIR*. In an ELF environment, ar does not use temporary files, and this option is ignored.

s Force the regeneration of the archive symbol table even if `ar` is not invoked with a command which will modify the archive contents. This command is useful to restore the archive symbol table after `strip`(1) or (`mcs`(1) has been used on the archive. This key can be used only in combination with one of the keys [`drqtpmx`].

FILES

`$TMPDIR/*` temporary files

`$TMPDIR` is usually `/usr/tmp` but can be redefined by setting the environment variable TMPDIR [see `tempnam`() in `tmpnam`(3S)]. In an ELF environment, ar no longer uses temporary files.

SEE ALSO

`ld`(1), `lorder`(1), `strip`(1), `mcs`(1), `a.out`(4), `ar`(4).

NOTES

By convention, archives are suffixed with the characters `.a`.

If the same file is mentioned twice in an argument list, it may be put in the archive twice.

NAME

as – MC88000 assembler

SYNOPSIS

as [*options*] *file*

DESCRIPTION

The as command performs assembly of 88000 instruction mnemonics into object files. The assembler input language is described in Chapter 11 of *Programmer's Guide: ANSI C and Programming Support Tools*. The as command may optionally invoke the m4(1) macro processor and sifilter(1) before assembly. The as command reads input from *file*; if *file* is '–', as reads from stdin.

as supports the following options:

–o *objfile*
: Causes as to place its output in the specified *objfile*. If this option is not present, as places output in a file whose name is constructed from *file* by replacing a .s suffix, if present, with .o, otherwise by appending .o. If as takes its input from stdin, then the –o option must be supplied. The output file must be a file on which as can perform fseek(3S).

–m
: Causes as to process its input with the m4 macro processor before assembly.

–Y [md],*dir*
: Normally, as will invoke m4 with a command line of the form:

/bin/m4 /lib/cm4defs *file*

The –Y option changes the directory from which m4 is invoked and the directory in which cm4defs is found. Thus
–Y m,*dir* will invoke *dir*/m4 and include /lib/cm4defs;
–Y d,*dir* will invoke /bin/m4 and include *dir*/cm4defs;
–Y md,*dir* will invoke *dir*/m4 and include *dir*/cm4defs.

–W s,*sifilter-options*
: This option controls invocation of the silicon filter before assembly. Arguments to this option include on, which will unconditionally invoke sifilter with default options, and off, which prevents invocation of sifilter; any other arguments are passed as options to sifilter.

By default, as will not invoke sifilter.

–W c,*ctl-options*
: This option controls invocation of the COFF-to-legend translator after assembly. All arguments are passed as options to ctl.

By default, as will not invoke ctl.

FILES

/bin/ctl	COFF-to-legend translator, ctl(1)
/bin/sifilter	silicon filter, sifilter(1)

SEE ALSO

cc(1), ld(1), m4(1), nm(1), strip(1), ctl(1), sifilter(1), tmpnam(3S), a.out(4).

NOTES

If the `-m` (`m4` macro processor invocation) option is used, keywords for `m4` [see `m4`(1)] cannot be used as symbols (variables, functions, labels) in the input file since `m4` cannot determine which keywords are assembler symbols and which keywords are real `m4` macros.

Arithmetic expressions may have only one forward referenced symbol per expression.

Whenever possible, you should access the assembler through a compilation system interface program such as `cc`.

NAME

asa – interpret ASA carriage control characters

SYNOPSIS

asa [*files*]

DESCRIPTION

Asa interprets the output of FORTRAN programs that utilize ASA carriage control characters. It processes either the *files* whose names are given as arguments or the standard input if no file names are supplied. The first character of each line is assumed to be a control character; the meanings are:

' ' (blank) single new line before printing

0 double new line before printing

1 new page before printing

+ overprint previous line.

Lines beginning with characters other than the ones above are treated as if they began with ' '. The first character of the line is *not* printed and an appropriate diagnostic will appear on standard error. This program forces the first line of each input file to start on a new page.

To view correctly the output of FORTRAN programs which use ASA carriage control characters, asa could be used as a filter thus:

```
a.out | asa | lp
```

and the output, properly formatted and paginated, would be directed to the line printer. FORTRAN output sent to a file could be viewed by:

```
asa file
```

SEE ALSO

f77(1), fsplit(1), ratfor(1).

NAME

att_dump – dump parts of an object or object archive file

SYNOPSIS

att_dump [*options*] *files*

DESCRIPTION

The att_dump command dumps selected parts of each of its *file* arguments.

This command will accept object and archives of object files. It processes each file argument according to one or more options. These options are supported in both ELF and COFF environments:

-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 program header from ELF files; dump optional header from COFF files.

-h Dump section headers.

-s Dump section contents.

-r Dump relocation information.

-l Dump line number information.

-t Dump symbol table entries.

-c Dump the string table.

-L Dump dynamic linking information and static shared library information, if available.

These options are supported only in an ELF environment:

-C Dump decoded C++ symbol table names.

-T *index* or -T *index1,index2*
Dump only the indexed symbol table entry defined by *index* or a range of entries defined by *index1,index2*.

-V Print version information.

-u When reading a COFF object file, att_dump translates the file to ELF internally (this translation does not affect the file contents). This option controls how much translation occurs from COFF values to ELF. Normally (without -u), the COFF values are preserved as much as possible, showing the actual bytes in the file. If -u is used, att_dump updates the values and completes the internal translation, giving a consistent ELF view of the contents. Although the bytes displayed under this option might not match the file itself, they show how the file would look if it were converted to ELF. (See cof2elf(1) for more information.)

This option is supported only in a COFF environment:

-z *name* Dump line number entries for the named function.

The following *modifiers* are used in conjunction with the options listed above to modify their capabilities.

-n *name* Dump information pertaining only to the named entity. This modifier applies to -h, -s, -r, -l, and -t. When -n is used with -h or -s, the argument will be treated as the name of a section. When -n is used with -t or -r, the argument will be treated as the name of a symbol. For example, dump -t -n .text will dump the symbol table entry associated with the symbol whose name is .text, whereas dump -h -n .text will dump the section header information for the .text section.

-p Suppress printing of the headers.

-v Dump information in symbolic representation rather than numeric (e.g., C_STATIC instead of 0X02).

-d *index* Dump only the indexed section. In an ELF environment, you may specify a range of sections as

-d *start-index,end-index*

In a COFF environment, use the +d modifier to specify a range of sections:

-d *start-index* +d *end-index*

These modifiers are accepted only in a COFF environment:

+d *index* Dump the sections in the range ending with the indexed section. The range begins at the first section or at the section specified by the -d option.

-t *index* Dump only the indexed symbol table entry.

+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.

-z *name,number*
Dump the line number entry or range of line numbers starting at *number* for the named function.

+z *number* Dump the line number entries starting at either function *name* or *line number* specified by -z, up to *number* specified by +z.

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 att_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.

Although the command produces no output when invoked without options, it does serve to verify that a file is an object, executable, or archive of an object or executable.

SEE ALSO

a.out(4), ar(4).

NAME

cb – C program beautifier

SYNOPSIS

cb [-s] [-j] [-l *leng*] [*file* ...]

DESCRIPTION

cb reads C programs either from the files specified in its arguments or from the standard input and writes them on the standard output. Spacing and indentation display the structure of the code. Under default options, *cb* preserves all user new-lines.

Options are:

-s Formats the code to the style of Kernighan and Ritchie in *The C Programming Language*.

-j Puts split lines in the input back together.

-l *leng* Causes *cb* to split lines that are longer than *leng*.

International Features

cb can process characters from supplementary code sets as well as ASCII characters.

SEE ALSO

cc(1).
The C Programming Language by B. W. Kernighan and D. M. Ritchie.

BUGS

Punctuation hidden in preprocessor statements will cause indentation errors.

NAME

cc – C language compiler

SYNOPSIS

cc [*option*] *filename* ...

DESCRIPTION

The cc command is the interface to the C compilation system. The system conceptually consists of a preprocessor, compiler, optimizer, assembler, and link-editor. The cc command processes the supplied options and then executes the various tools with the appropriate arguments.

The gcc command accesses the GNU C compiler. For a further description see gcc(1). The ghcc command accesses the Green Hills C compiler; see ghcc(1). The Green Hills C compiler is a separate product and may not exist on your system.

The cc command invokes gcc with the `-traditional` option. This means that cc will attempt to support PCC features. Facilities unique to gcc may not be accessible from the cc command; instead you must use gcc directly.

The suffix of a *filename* argument indicates how the file is to be treated. Files whose names end with `.c` are taken to be C source programs and may be preprocessed, compiled, optimized, assembled, and link-edited. The compilation process may be stopped after the completion of any pass if the appropriate options are supplied. If the compilation process is allowed to complete the assembly phase, then an object file is produced; the object file for a source file called `xyz.c` is created in a file called `xyz.o`. However, the `.o` file is normally deleted if a single C program is compiled and loaded all at one go.

In the same way, arguments whose names end with `.s` are taken to be assembly source programs, and may be assembled and link-edited. Files with names ending in `.i` are taken to be preprocessed C source programs and may be compiled, optimized, assembled, and link-edited. Files whose names do not end in `.c`, `.s`, or `.i` are handed to the link-editor.

By default, if an executable file is produced (i.e., the link-edit phase is allowed to finish), the file is called `a.out`. This default name can be changed with the `-o` option (see below).

Options

Some options to cc are sensitive to the sde target environment (see sde(5), `sde-target`(1)). Options unique to ELF or COFF target environments are so indicated in the following list.

These options are interpreted by cc:

-ansi Compile the source in accordance with rules for ANSI C and flag violations (this is equivalent to the `-Xc` option). `Cc -ansi` has the same effect as `gcc -ansi -pedantic`.

-X [tac]
Specify the degree of conformance to the ANSI C standard. The arguments have the following meanings:

t (transition)
The compiled language includes all new features compatible with older (pre-ANSI) C (the default behavior). The compiler warns about all language constructs that have differing behavior between the new and old versions and uses the pre-ANSI C interpretation. This includes, for example, warning about the use of trigraphs the new escape sequence `\a`, and

the changes to the integral promotion rules. `cc -Xt` has the same effect as `gcc -traditional`.

a (ANSI)
The compiled language includes all new features of ANSI C and uses the new interpretation of constructs with differing behavior. The compiler continues to warn about the integral promotion rule changes, but does not warn about new escape sequences.

c (conformance)
The compiled language and associated header files are ANSI C conforming, but include all conforming extensions of `-Xa`. Warnings will be produced about some of these. Also, only ANSI defined identifiers are visible in the standard header files. (This is equivalent to the `-ansi` option.)

The predefined macro `__STDC__` has the value `1` for `-Xa` and `-Xc`. All warning messages about differing behavior can be eliminated in `-Xa` through appropriate coding; for example, use of casts can eliminate the integral promotion change warnings.

These options also affect the behavior of libc and libm routines if present on the command line at link time.

`-O` Do compilation-phase optimization on `.c` or `.i` files. This option will not affect code produced from `.s` files.

`-O2` Do aggressive compilation-phase optimization on `.c` or `.i` files. All supported optimizations are performed. As compared to `-O`, this option will increase both compilation time and the performance of the generated code. The `-O2` option is supported only by Version 2 of the GNU C compiler (see the `-K V` option, below).

`-g` Cause the compiler to generate additional information needed for the use of a debugger.

`-p` Arrange for the compiler to produce code that counts the number of times each routine is called: also, if link-editing takes place, a profiled version of the standard C library is linked, and `monitor` (see `monitor`(3C)) is automatically called. A `mon.out` file will then be produced on normal termination of the program. An execution profile can then be generated by use of `prof`. Default parameters to `monitor` ensure that up to 600 call counts are captured and that each pc has a corresponding histogram bucket in the `mon.out` file.

`-D` *name*[=*tokens*]
Associate name with the specified tokens as if by a `#define` preprocessor directive. If no =*tokens* is specified, the token `1` is supplied.

`-U` *name*
Cause any definition of *name* to be forgotten, as if by a `#undef` preprocessor directive. If the same name is specified for both `-D` and `-U`, *name* is not defined, regardless of the order of the options.

`-V` Cause each invoked tool to print its version information on the standard error output.

`-v` Print the invocation of each tool on the standard error output.

`-K` [`PIC` [,V*version*]]

-K PIC (ELF only)
Generate position-independent code (PIC).

-K V*version*
Select a version of the GNU C compiler. The command cc -KV lists *versions* available on the system. (The command default-gcc is used to determine or to change the system default.)

The -K option can accept multiple arguments. For example, -K PIC,V2 can be used instead of -K PIC -K V2.

-E Preprocess the named C programs and send the result to the standard output.

-P Preprocess the named C programs and leave the result in corresponding files suffixed .i.

-S Compile and do not assemble or link-edit the named C files. The assembly language output is left in corresponding files suffixed .s.

-C Cause the preprocessing phase to pass along all comments other than those on preprocessing directive lines.

-H Cause pathnames of files included during preprocessing to be printed on the standard error output.

-c Suppress the link edit phase of the compilation, and do not remove any object files produced.

-o *outfile*
Use the name *outfile*, instead of the default a.out, for the executable file produced. This is a link-editor option and does not apply to files produced by the -S, -c, or -P options.

-d [y | n] (ELF only)
-dy specifies dynamic linking, which is the default, in the link editor. -dn specifies static linking in the link editor. This option and its argument are passed to ld.

-G (ELF only)
Direct the link editor to produce a shared object rather than a dynamically linked executable. This option cannot be used with the -dn option.

-B [dynamic | static] (ELF only)
-B dynamic causes the link editor to look for files named lib*x*.so and then for files named lib*x*.a when given the -l*x* option. -B static causes the link editor to look only for files named lib*x*.a. These options may be specified multiple times on the command line as a toggle.

-B symbolic (ELF only)
Direct the link editor to bind references to global symbols to their definitions within the object, if definitions are available, when building a shared object. This option is meaningful only in dymnamic mode.

The -B option and its argument are passed to the link editor.

-B *string* (COFF only)
Construct pathnames for substitute preprocessor, compiler, optimizer, assembler, COFF-to-legend translator, and link-editor passes by concatenating *string* with the appropriate suffix. If *stringfl is empty it is taken to be* /lib/o. This option is obsolete; -Y should be used instead.

-Q [y | n] (ELF only)
: -Qy directs the link editor to add identification information to the output file (the default behavior); this can be useful for software administration. -Qn suppresses this information.

-I *dir* Alter the search for included files whose names do not begin with / to look in *dir* prior to the usual directories. The directories for multiple -I options are searched in the order specified.

-L *dir* Add *dir* to the list of directories searched for libraries by ld. This option and its argument are passed to the link editor.

-l *name*
: Search the library lib*name*.so or lib*name*.a. Its placement on the command line is significant as a library is searched at a point in time relative to the placement of other libraries and object files on the command line. This option and its argument are passed to the link editor.

-W *c,arg1[,arg2...]*
: Hand off the argument(s) *argi* to phase *c* where *c* is one of [p02sacl] indicating preprocessing, compilation, optimization, assembly, COFF-to-legend symbol-table translation, or link-editing phases, respectively. For example, -W a,-m passes -m to the assembler phase.

-Y *items,dir*
: Specify a new directory *dir* for the location of the tools and directories designated in the first argument. *items* can consist of any grouping of the following characters:

p	preprocessor
0	compiler
2	optimizer
a	assembler
c	COFF-to-legend translator
l	link-editor
I	directory searched last for include files (default /usr/include)
L	directory searched next to last for libraries (default /usr/lib)
S	directory containing the start-up object files (default /usr/lib)
U	directory searched last for libraries (default /usr/lib)

If the location of a tool is being specified, then the new pathname for the tool will be *dir/tool*. If more than one -Y option is applied to any one item, the last occurrence holds.

-t *items* (COFF only)
: Find only the tools designated by *items* in the file whose name is constructed by a -Y option. In the absence of a -Y option, the prefix is taken to be /lib/n. *items* can be zero or more letters from [p02sacl], designating the preprocessor, compiler, optimizer, assembler, COFF-to-legend translator, or link-editor. If *items* is empty (as in '-t""'), all tools are designated.

The cc command passes any unrecognized options to ld without any diagnostic (see ld(1) for descriptions of ld options).

Other arguments are taken to be C-compatible object programs or libraries of C-compatible routines and are passed directly to the link-editor. These programs, together with the results of any compilations specified, are linked (in the order given) to produce an executable program with the name `a.out` (unless the `-o` link-editor option is used).

The standard C library is automatically available to the C program. Other libraries must be specified explicitly using the `-l` option with `cc` (see `ld`(1) for details).

#define Statements

The following list provides the meaning of symbols that are defined by default under `cc`. When defined, the value is 1.

`__m88k__`
: The target system is a Motorola 88100.

`__unix__`
: Unix operating system.

`__DGUX__`
: DG/UX operating system.

`__GNUC__`
: Defined as `1` or `2` by version 1 or 2 of the GNU C compiler.

`__STDC__`
: ANSI features are assumed. Defined when `-ansi`, `-Xa` or `-Xc` is given.

`__STRICT_ANSI__`
: Strict ANSI, no extensions. Defined when `-ansi` or `-Xc` is given.

`__CLASSIFY_TYPE__`
: Defined as `1` or `2` by version 1 or 2 of the GNU C compiler; selects the `varargs` method of the respective compiler.

`__OPEN_NAMESPACE__`
: Defined when `-Xa` is given. Non-ANSI C standard features in header files are visible during compilation.

Additionally, when the compiler is not in strict ANSI mode (ANSI prohibits predefined names that don't begin with either two '_'s, or an '_' and an uppercase letter) the following are also available:

`m88000` Deprecated alternative of `__m88k__`.

`m88k` Deprecated alternative of `__m88k__`.

`unix` Deprecated alternative of `__unix__`.

`DGUX` Deprecated alternative of `__DGUX__`.

There are several macros you can define to control your source and target environments when developing applications. These macros control header files, function declarations, binary formats, and other aspects of the source and target environments. The macros are helpful when you are porting applications to or from non-DG/UX systems such as BSD or AT&T systems. The macros can also make development of POSIX- or BCS-conformant applications easier. For developing BCS-conformant applications, the `sde` utility is also helpful. See *Porting Applications to the DG/UX™ System* and the `sde-target`(1), `sdetab`(4), and `sde`(5) manual pages.

FILES

`file.c`	C source file
`file.i`	preprocessed C source file
`file.o`	object file
`file.s`	assembly language file
`a.out`	link-edited output
`$TMPDIR/ctm*`	temporary files. `$TMPDIR` is usually `/tmp` but can be redefined by setting the environment variable `TMPDIR`.
`/usr/lib/gcc/gcc-cpp`	GNU preprocessor
`/usr/lib/gcc/gcc-cc1`	GNU C compiler
`/bin/as`	assembler, `as`(1)
`/bin/ld`	link editor, `ld`(1)
`/bin/ctl`	COFF-to-legend translator, `ctl`(1)
`/lib/crt0.o`	start-up routine
`/lib/mcrt0.o`	profiling start-up routine
`/lib/libc.a`	standard C library

SEE ALSO

`as`(1), `ctl`(1), `gcc`(1), `ld`(1), `sde-target`(1), `sdetab`(4), `sde`(5).

NOTES

The `-f` option is ignored on 88000 systems. Floating-point support is always present.

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* specified by the -r option of each named *SID* file.

Delta commentary is defined to be the Modification Request (MR) and comment information normally specified via the delta(1) command (-m and -y options).

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 pathname 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.

Arguments to cdc can appear in any order. They consist of options and filenames.

All the described options apply independently to each named file:

-r*SID* Specifies the *S*CCS *ID*entification (ID) *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 you can supply a list of MR numbers to be added and/or deleted in the delta commentary of the *SID* specified by the -r option. A null MR list has no effect. MR entries are added to the list of MRs as in delta(1). 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 option). 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) that validates 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 that replaces the current *comment*(s) for the delta specified by the -r option. 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.

The exact permissions necessary to modify the SCCS file are documented in *Programmer's Guide: ANSI C and Programming Support Tools*. Simply stated, they 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

$ cdc −r1.6 s.file
MRs? **!bl77-54321 bl78-12345 bl79-00001**
comments? **trouble**

Both examples add `bl78-12345` and `bl79-00001` to the MR list, remove `bl77-54321` from the MR list, and add the comment `trouble` to delta 1.6 of `s.file`.

FILES

`x-file` (see `delta`(1))
`z-file` (see `delta`(1))

DIAGNOSTICS

Use `help`(1) for explanations of error messages.

SEE ALSO

`admin`(1), `comb`(1), `delta`(1), `get`(1), `help`(1), `prs`(1), `sccsfile`(4).

NOTES

If SCCS filenames are supplied to the `cdc` command via the standard input (− on the command line), then the `−m` and `−y` options must also be used.

NAME

cflow – generate a C flow graph

SYNOPSIS

cflow [-r] [-ix] [-i_] [-D*name=value*] [-U*name*] [-I*dir*] [-d*num*] *filename* ...

DESCRIPTION

Cflow analyzes a collection of C, yacc, lex, assembler, and object files and builds a graph charting the external function references. Files suffixed with .y, .l, and .c are processed by yacc, lex, and the C compiler as appropriate. The results of the preprocessed files, and files suffixed with .i, are then run through the first pass of lint. 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 that is written on the standard output.

Each line of output begins with a line 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, use the -e option of pr(1) 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), 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).

-d*num* The *num* decimal integer indicates the depth at which the flowgraph is cut off. By default this number is very large. Attempts to set the cutoff depth to a nonpositive integer will be ignored.

DIAGNOSTICS

Complains about bad options. Complains about multiple definitions and believes only 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), pr(1), yacc(1).

NOTES

Files produced by lex(1) and yacc(1) reorder line number declarations, which can confuse cflow. To get proper results, feed cflow the yacc or lex input.

NAME

ci - check in RCS revisions

SYNOPSIS

ci [*options*] *file* ...

DESCRIPTION

ci stores new revisions into RCS files. Each file name ending in ',v' is taken to be an RCS file, all others are assumed to be working files containing new revisions. ci deposits the contents of each working file into the corresponding RCS file.

Pairs of RCS files and working files may be specified in 3 ways (see also the example section of co(1)).

1) Both the RCS file and the working file are given. The RCS file name is of the form *path1/workfile*,v and the working file name is of the form *path2/workfile*, where *path1/* and *path2/* are (possibly different or empty) paths and *workfile* is a file name.

2) Only the RCS file is given. Then the working file is assumed to be in the current directory and its name is derived from the name of the RCS file by removing *path1/* and the suffix ',v'.

3) Only the working file is given. Then the name of the RCS file is derived from the name of the working file by removing *path2/* and appending the suffix ',v'.

If the RCS file is omitted or specified without a path, then ci looks for the RCS file first in the directory ./RCS and then in the current directory.

For ci to work, the caller's login must be on the access list, except if the access list is empty or the caller is the superuser or the owner of the file. To append a new revision to an existing branch, the tip revision on that branch must be locked by the caller. Otherwise, only a new branch can be created. This restriction is not enforced for the owner of the file, unless locking is set to strict (see rcs(1)). A lock held by someone else may be broken with the rcs command.

Normally, ci checks whether the revision to be deposited is different from the preceding one. If it is not different, ci either aborts the deposit (if -q is given) or asks whether to abort (if -q is omitted). A deposit can be forced with the -f option.

For each revision deposited, ci prompts for a log message. The log message should summarize the change and must be terminated with a line containing a single '.' or a control-D. If several files are checked in, ci asks whether to reuse the previous log message. If the std. input is not a terminal, ci suppresses the prompt and uses the same log message for all files. See also -m.

The number of the deposited revision can be given by any of the options -r, -f, -k, -l, -u, -q or -c (see -r).

If the RCS file does not exist, ci creates it and deposits the contents of the working file as the initial revision (default number: 1.1). The access list is initialized to empty. Instead of the log message, ci requests descriptive text (see -t below).

-r[*rev*] assigns the revision number *rev* to the checked-in revision, releases the corresponding lock, and deletes the working file. This is also the default.

If *rev* is omitted, ci derives the new revision number from the caller's last lock. If the caller has locked the tip revision of a branch, the new revision is appended to that branch. The new revision number is obtained by incrementing the tip revision number. If the caller locked a non-tip revision, a new branch is started at that revision by incrementing the highest branch number at that revision. The default initial branch and level numbers are 1. If the caller holds no lock, but he is the owner of the file and locking is not set to `strict`, then the revision is appended to the trunk.

If *rev* indicates a revision number, it must be higher than the latest one on the branch to which *rev* belongs, or must start a new branch.

If *rev* indicates a branch instead of a revision, the new revision is appended to that branch. The level number is obtained by incrementing the tip revision number of that branch. If *rev* indicates a non-existing branch, that branch is created with the initial revision numbered *rev.1*.

Exception: On the trunk, revisions can be appended to the end, but not inserted.

`-f`[*rev*] forces a deposit; the new revision is deposited even if it is not different from the preceding one.

`-k`[*rev*] searches the working file for keyword values to determine its revision number, creation date, author, and state (see `co`(1)), and assigns these values to the deposited revision, rather than computing them locally. A revision number given by a command option overrides the number in the working file. This option is useful for software distribution. A revision that is sent to several sites should be checked in with the `-k` option at these sites to preserve its original number, date, author, and state.

`-l`[*rev*] works like `-r`, except it performs an additional `co -l` for the deposited revision. Thus, the deposited revision is immediately checked out again and locked. This is useful for saving a revision although one wants to continue editing it after the checkin.

`-u`[*rev*] works like `-l`, except that the deposited revision is not locked. This is useful if one wants to process (e.g., compile) the revision immediately after checkin.

`-q`[*rev*] quiet mode; diagnostic output is not printed. A revision that is not different from the preceding one is not deposited, unless `-f` is given.

`-c`[*rev*] no changes mode; the working file is assumed to be unchanged. An unchanged revision is deposited without the overhead of determining what changes have been made.

`-m`*msg* uses the string *msg* as the log message for all revisions checked in.

`-n`*name* assigns the symbolic name *name* to the number of the checked-in revision. `ci` prints an error message if *name* is already assigned to another number. Names must begin with a letter, and cannot contain whitespace, period, colon, semicolon, or @.

`-N`*name* same as `-n`, except that it overrides a previous assignment of *name*.

`-s`*state* sets the state of the checked-in revision to the identifier *state*. The default is `Exp`. Any string that could be a name (see `-n`) is acceptable for *state*.

`-t`[*txtfile*] writes descriptive text into the RCS file (deletes the existing text). If *txtfile* is omitted, `ci` prompts the user for text supplied from the std. input, terminated with a line containing a single '.' or control-D. Otherwise, the descriptive text is copied from the file *txtfile*. During initialization, descriptive text is requested even if `-t` is not given. The prompt is suppressed if std. input is not a terminal.

File Modes

An RCS file created by `ci` inherits the read and execute permissions from the working file. If the RCS file exists already, `ci` preserves its read and execute permissions. `ci` always turns off all write permissions of RCS files.

The caller of the command must have read/write permission for the directories containing the RCS file and the working file, and read permission for the RCS file itself.

DIAGNOSTICS

For each revision, `ci` prints the RCS file, the working file, and the number of both the deposited and the preceding revision. The exit status always refers to the last file checked in, and is 0 if the operation was successful, 1 otherwise.

FILES

A number of temporary files are created. A semaphore file is created in the directory containing the RCS file. `ci` always creates a new RCS file and unlinks the old one. This strategy makes links to RCS files useless.

SEE ALSO

`co`(1), `ident`(1), `rcs`(1), `rcsdiff`(1), `rcsintro`(1), `rcsmerge`(1), `rlog`(1), `rcsfile`(4), `sccstorcs`(8).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering,* IEEE, Tokyo, Sept. 1982.

NAME

ckdate, errdate, helpdate, valdate – prompt for and validate a date

SYNOPSIS

ckdate [-Q] [-W *width*] [-f *format*] [-d *default*] [-h *help*] [-e *error*] [-p *prompt*] [-k *pid* [-s *signal*]]

/usr/sadm/bin/errdate [-W] [-e *error*] [-f *format*]
/usr/sadm/bin/helpdate [-W] [-h *help*] [-f *format*]
/usr/sadm/bin/valdate [-f *format]* *input*

DESCRIPTION

Ckdate prompts a user and validates the response. It defines, among other things, a prompt message whose response should be a date, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return). The user response must match the defined format for a date.

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The -W option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the ckdate command. They are errdate (which formats and displays an error message), helpdate (which formats and displays a help message), and valdate (which validates a response). These modules should be used in conjunction with FML objects. In this instance, the FML object defines the prompt. When *format* is defined in the errdate and helpdate modules, the messages will describe the expected format.

The options and arguments for this command are:

-Q Specifies that quit will not be allowed as a valid response.

-W *width* Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-f *format* Specifies the format against which the input will be verified. Possible formats and their definitions are:
%b = abbreviated month name
%B = full month name
%d = day of month (01 - 31)
%D = date as %m/%d/%y (the default format)
%e = day of month (1 - 31; single digits are preceded by a blank)
%h = abbreviated month name (jan, feb, mar)
%m = month number (01 - 12)
%y = year within century (e.g. 89)
%Y = year as CCYY (e.g. 1989)

-d *default* Defines the default value as *default*.
The default does not have to meet the format criteria.

-h *help* Defines the help messages as *help*.

-e *error* Defines the error message as *error*.

-p *prompt* Defines the prompt message as *prompt*.

-k *pid* Specifies that process ID *pid* is to be sent a signal if the user chooses to abort.

`-s` *signal* Specifies that the process ID *pid* defined with the `-k` option is to be sent signal `signal` when quit is chosen. If no signal is specified, `SIGTERM` is used.

input Input to be verified against format criteria.

EXIT CODES

0 = Successful execution
1 = EOF on input
2 = Usage error
3 = User termination (quit)
4 = Garbled format argument

SEE ALSO

`valtools`(1).

NOTES

The default prompt for `ckdate` is:

```
Enter the date [?,q]:
```

The default error message is:

`ERROR - Please enter a date, using the following format:` *format*.

The default help message is:

`Please enter a date, using the following format:` *format*.

When the quit option is chosen (and allowed), `q` is returned along with the return code `3`. The `valdate` module will not produce any output. It returns zero for success and non-zero for failure.

NAME

`ckgid`, `errgid`, `helpgid`, `valgid` – prompt for and validate a group id

SYNOPSIS

`ckgid` [`-Q`] [`-W` *width*] [`-m`] [`-d` *default*] [`-h` *help*] [`-e` *error*] [`-p` *prompt*] [`-k` *pid* [`-s` *signal*]]

`/usr/sadm/bin/errgid` [`-W`] [`-e` *error*]
`/usr/sadm/bin/helpgid` [`-W`] [`-m`] [`-h` *help*]
`/usr/sadm/bin/valgid` *input*

DESCRIPTION

`ckgid` prompts a user and validates the response. It defines, among other things, a prompt message whose response should be an existing group ID, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The `-W` option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the `ckgid` command. They are `errgid` (which formats and displays an error message), `helpgid` (which formats and displays a help message), and `valgid` (which validates a response). These modules should be used in conjunction with FML objects. In this instance, the FML object defines the prompt.

The options and arguments for this command are:

`-Q`	Specifies that quit will not be allowed as a valid response.
`-W` *width*	Specifies that prompt, help and error messages will be formatted to a line length of *width*.
`-m`	Displays a list of all groups when help is requested or when the user makes an error.
`-d` *default*	Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.
`-h` *help*	Defines the help messages as *help*.
`-e` *error*	Defines the error message as *error*.
`-p` *prompt*	Defines the prompt message as *prompt*.
`-k` *pid*	Specifies that process ID *pid* is to be sent a signal if the user chooses to abort.
`-s` *signal*	Specifies that the process ID *pid* defined with the `-k` option is to be sent signal `signal` when quit is chosen. If no signal is specified, `SIGTERM` is used.
input	Input to be verified against `/etc/group`

EXIT CODES

0 = Successful execution
1 = EOF on input
2 = Usage error
3 = User termination (quit)

SEE ALSO

valtools(1).

NOTES

The default prompt for ckgid is:

`Enter the name of an existing group [?,q]:`

The default error message is:

`ERROR - Please enter the name of an existing group.`
(if the `-m` *option of* `ckgid` *is used, a list of valid groups is displayed here)*

The default help message is:

`Please enter an existing group name.`
(if the `-m` *option of* `ckgid` *is used, a list of valid groups is displayed here)*

When the quit option is chosen (and allowed), `q` is returned along with the return code 3. The `valgid` module will not produce any output. It returns zero for success and non-zero for failure.

NAME

`ckint` – display a prompt; verify and return an integer value

SYNOPSIS

`ckint` [`-Q`] [`-W` *width*] [`-b` *base*] [`-d` *default*] [`-h` *help*] [`-e` *error*] [`-p` *prompt*] [`-k` *pid* [`-s signal`]]

`/usr/sadm/bin/errint` [`-W`] [`-b` *base*] [`-e` *error*]
`/usr/sadm/bin/helpint` [`-W`] [`-b` *base*] [`-h` *help*]
`/usr/sadm/bin/valint` [`-b` *base*] *input*

DESCRIPTION

`ckint` prompts a user, then validates the response. It defines, among other things, a prompt message whose response should be an integer, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The `-W` option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the `ckint` command. They are `errint` (which formats and displays an error message), `helpint` (which formats and displays a help message), and `valint` (which validates a response). These modules should be used in conjunction with FML objects. In this instance, the FML object defines the prompt. When *base* is defined in the `errint` and `helpint` modules, the messages will include the expected base of the input.

The options and arguments for this command are:

- `-Q` Specifies that quit will not be allowed as a valid response.
- `-W` Specifies that prompt, help and error messages will be formatted to a line length of *width*.
- `-b` Defines the base for input. Must be 2 to 36, default is 10.
- `-d` Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.
- `-h` Defines the help messages as *help*.
- `-e` Defines the error message as *error*.
- `-p` Defines the prompt message as *prompt*.
- `-k` Specifies that process ID *pid* is to be sent a signal if the user chooses to abort.
- `-s` Specifies that the process ID *pid* defined with the `-k` option is to be sent signal `signal` when quit is chosen. If no signal is specified, `SIGTERM` is used.
- *input* Input to be verified against *base* criterion.

EXIT CODES

0 = Successful execution
1 = EOF on input
2 = Usage error

3 = User termination (quit)

SEE ALSO

`valtools`(1).

NOTES

The default base 10 prompt for `ckint` is:

```
Enter an integer [?,q]:
```

The default base 10 error message is:

```
ERROR - Please enter an integer.
```

The default base 10 help message is:

```
Please enter an integer.
```

The messages are changed from "integer" to "base *base* integer" if the base is set to a number other than 10.

When the quit option is chosen (and allowed), `q` is returned along with the return code 3. The `valint` module will not produce any output. It returns zero for success and non-zero for failure.

NAME

`ckitem` – build a menu; prompt for and return a menu item

SYNOPSIS

`ckitem` [`-Q`] [`-W` *width*] [`-uno`] [`-f` *file*] [`-l` *label*] [[`-i` *invis*] [, ...]] [`-m` *max*] [`-d` *default*] [`-h` *help*] [`-e` *error*] [`-p` *prompt*] [`-k` *pid* [`-s` *signal*]] [*choice* [...]]

`/usr/sadm/bin/erritem` [`-W`] [`-e` *error*] [*choice* [...]]
`/usr/sadm/bin/helpitem` [`-W`] [`-h` *help*] [*choice* [...]]

DESCRIPTION

`ckitem` builds a menu and prompts the user to choose one item from a menu of items. It then verifies the response. Options for this command define, among other things, a prompt message whose response will be a menu item, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

By default, the menu is formatted so that each item is prepended by a number and is printed in columns across the terminal. Column length is determined by the longest choice. Items are alphabetized.

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The `-W` option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Two visual tool modules are linked to the `ckitem` command. They are `erritem` (which formats and displays an error message) and `helpitem` (which formats and displays a help message). These modules should be used in conjunction with FML objects. In this instance, the FML object defines the prompt. When *choice* is defined in these modules, the messages will describe the available menu choice (or choices).

The options and arguments for this command are:

`-Q` Specifies that quit will not be allowed as a valid response.

`-W` Specifies that prompt, help and error messages will be formatted to a line length of *width*.

`-u` Specifies that menu items should be displayed as an unnumbered list.

`-n` Specifies that menu items should not be displayed in alphabetical order.

`-o` Specifies that only one menu token will be returned.

`-f` Defines a file, *file*. which contains a list of menu items to be displayed. [The format of this file is: `token`*tab*description. Lines beginning with a pound sign (#) are designated as comments and ignored.]

`-l` Defines a label, *label*, to print above the menu.

`-i` Defines invisible menu choices (those which will not be printed in the menu). (For example, "all" used as an invisible choice would mean it is a legal option but does not appear in the menu. Any number of invisible choices may be defined.) Invisible choices should be made known to a user either in the prompt or in a help message.

-m Defines the maximum number of menu choices allowed.

-d Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

-h Defines the help messages as *help*.

-e Defines the error message as *error*.

-p Defines the prompt message as *prompt*.

-k Specifies that the process ID *pid* is to be sent a signal if the user chooses to abort.

-s Specifies that process ID *pid* defined with the -k option is to be sent signal signal when quit is chosen. If no signal is specified, SIGTERM is used.

choice Defines menu items. Items should be separated by white space or newline.

EXIT CODES

0 = Successful execution
1 = EOF on input
2 = Usage error
3 = User termination (quit)
4 = No choices from which to choose

SEE ALSO

valtools(1).

NOTES

The user may input the number of the menu item if choices are numbered or as much of the string required for a unique identification of the item. Long menus are paged with 10 items per page.

When menu entries are defined both in a file (by using the -f option) and also on the command line, they are usually combined alphabetically. However, if the -n option is used to suppress alphabetical ordering, then the entries defined in the file are shown first, followed by the options defined on the command line.

The default prompt for ckitem is:

```
Enter selection [?,??,q]:
```

One question mark will give a help message and then redisplay the prompt. Two question marks will give a help message and then redisplay the menu label, the menu and the prompt.

The default error message is:

```
ERROR - Does not match an available menu selection.
Enter one of the following:
  the number of the menu item you wish to select
  the token associated withe the menu item,
  partial string which uniquely identifies the token for the
menu item
  ?? to reprint the menu
```

The default help message is:

```
Enter one of the following:
```

```
    the number of the menu item you wish to select
    the token associated with the menu item,
    partial string which uniquely identifies the token for the
  menu item
    ?? to reprint the menu
```

When the quit option is chosen (and allowed), q is returned along with the return code 3.

NAME

ckkeywd – prompt for and validate a keyword

SYNOPSIS

ckkeywd [-Q] [-W *width*] [-d *default*] [-h *help*] [-e *error*] [-p *prompt*]
[-k *pid* [-s *signal*]] [*keyword* [...]]

DESCRIPTION

ckkeywd prompts a user and validates the response. It defines, among other things, a prompt message whose response should be one of a list of keywords, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return). The answer returned from this command must match one of the defined list of keywords.

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The -W option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

-Q Specifies that quit will not be allowed as a valid response.

-W Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-d Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

-h Defines the help messages as *help*.

-e Defines the error message as *error*.

-p Defines the prompt message as *prompt*.

-k Specifies that process ID *pid* is to be sent a signal if the user chooses to abort.

-s Specifies that the process ID *pid* defined with the -k option is to be sent signal signal when quit is chosen. If no signal is specified, SIGTERM is used.

keyword
Defines the keyword, or list of keywords, against which the answer will be verified.

EXIT CODES

0 = Successful execution
1 = EOF on input
2 = Usage error
3 = User termination (quit)
4 = No keywords from which to choose

SEE ALSO

valtools(1).

NOTES

The default prompt for ckkeywd is:

```
Enter selection [keyword,[...],?,q]:
```

The default error message is:

```
ERROR - Does not match any of the valid selections.
Please enter one of the following keywords:
keyword[,...]
```

The default help message is:

```
Please enter one of the following keywords:
keyword[,...]
```

When the quit option is chosen (and allowed), `q` is returned along with the return code `3`.

NAME

ckpath – display a prompt; verify and return a pathname

SYNOPSIS

ckpath [-Q] [-W *width*] [-a|l] [-b|c|g|y] [-n|[o|z]] [-rtwx] [-d *default*] [-h *help*] [-e *error*] [-p *prompt*] [-k *pid* [-s *signal*]]

/usr/sadm/bin/errpath [-W] [-a|l] [-b|c|g|y] [-n|[o|z]] [-rtwx] [-e *error*]
/usr/sadm/bin/helppath [-W] [-a|l] [-b|c|g|y] [-n|[o|z]] [-rtwx] [-h *help*]
/usr/sadm/bin/valpath [-a|l] [-b|c|g|y] [-n|[o|z]] [-rtwx] *input*

DESCRIPTION

ckpath prompts a user and validates the response. It defines, among other things, a prompt message whose response should be a pathname, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

The pathname must obey the criteria specified by the first group of options. If no criteria is defined, the pathname must be for a normal file that does not yet exist. If neither -a (absolute) or -l (relative) is given, then either is assumed to be valid.

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The -W option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the ckpath command. They are errpath (which formats and displays an error message), helppath (which formats and displays a help message), and valpath (which validates a response). These modules should be used in conjunction with FACE objects. In this instance, the FACE object defines the prompt.

The options and arguments for this command are:

- -Q Specifies that quit will not be allowed as a valid response.
- -W Specifies that prompt, help and error messages will be formatted to a line length of *width*.
- -a Pathname must be an absolute path.
- -l Pathname must be a relative path.
- -b Pathname must be a block special file.
- -c Pathname must be a character special file.
- -g Pathname must be a regular file.
- -y Pathname must be a directory.
- -n Pathname must not exist (must be new).
- -o Pathname must exist (must be old).
- -z Pathname must have a length greater than 0 bytes.
- -r Pathname must be readable.
- -t Pathname must be creatable (touchable). Pathname will be created if it does not already exist.

−w Pathname must be writable.

−x Pathname must be executable.

−d Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

−h Defines the help messages as *help*.

−e Defines the error message as *error*.

−p Defines the prompt message as *prompt*.

−k Specifies that process ID *pid* is to be sent a signal if the user chooses to abort.

−s Specifies that the process ID *pid* defined with the `-k` option is to be sent signal `signal` when quit is chosen. If no signal is specified, `SIGTERM` is used.

input Input to be verified against validation options.

EXIT CODES

0 = Successful execution
1 = EOF on input
2 = Usage error
3 = User termination (quit)
4 = Mutually exclusive options

SEE ALSO

`valtools`(1).

NOTES

The text of the default messages for `ckpath` depends upon the criteria options that have been used. An example default prompt for `ckpath` (using the `-a` option) is:

```
Enter a pathname [?,q]:
```

An example default error message (using the `-a` option) is:

```
ERROR - Invalid pathname entered. A pathname is a filename,
optionally preceded by parent directories.
```

An example default help message is:

```
A pathname is a filename, optionally preceded by parent direc-
tories. The pathname you enter:
  must contain 1 to {NAME_MAX} characters
  must not contain a spaces or special characters
```

NAME_MAX is a system variable that is defined in `limits.h`.

When the quit option is chosen (and allowed), `q` is returned along with the return code `3`. The `valpath` module will not produce any output. It returns zero for success and non-zero for failure.

NAME

ckrange – prompt for and validate an integer

SYNOPSIS

ckrange [-Q] [-W *width*] [-l *lower*] [-u *upper*] [-b *base*] [-d *default*] [-h *help*] [-e *error*] [-p *prompt*] [-k *pid* [-s *signal*]]

/usr/sadm/bin/errange [-W] [-l *lower*] [-u *upper*] [-e *error*]
/usr/sadm/bin/helprange [-W] [-l *lower*] [-u *upper*] [-h *help*]
/usr/sadm/bin/valrange [-l *lower*] [-u *upper*] [-b *base*] *input*

DESCRIPTION

ckrange prompts a user and validates the response. It defines, among other things, a prompt message whose response should be an integer in the range specified, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

This command also defines a range for valid input. If either the lower or upper limit is left undefined, then the range is bounded on only one end.

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The -W option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the ckrange command. They are errange (which formats and displays an error message), helprange (which formats and displays a help message), and valrange (which validates a response). These modules should be used in conjunction with FACE objects. In this instance, the FACE object defines the prompt.

The options and arguments for this command are:

- -Q Specifies that quit will not be allowed as a valid response.
- -W Specifies that prompt, help and error messages will be formatted to a line length of *width*.
- -l Defines the lower limit of the range as *lower*. Default is the machine's largest negative integer or long.
- -u Defines the upper limit of the range as *upper*. Default is the machine's largest positive integer or long.
- -b Defines the base for input. Must be 2 to 36, default is 10.
- -d Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.
- -h Defines the help messages as *help*.
- -e Defines the error message as *error*.
- -p Defines the prompt message as *prompt*.
- -k Specifies that process ID *pid* is to be sent a signal if the user chooses to abort.

-s Specifies that the process ID *pid* defined with the -k option is to be sent signal signal when quit is chosen. If no signal is specified, SIGTERM is used.

input Input to be verified against upper and lower limits and base.

EXIT CODES

0 = Successful execution
1 = EOF on input
2 = Usage error
3 = User termination (quit)

SEE ALSO

valtools(1).

NOTES

The default base 10 prompt for ckrange is:

Enter an integer between *lower_bound* and *upper_bound* [q,?]:

The default base 10 error message is:

ERROR - Please enter an integer between *lower_bound* and *upper_bound*.

The default base 10 help message is:

Please enter an integer between *lower_bound* and *upper_bound*.

The messages are changed from "integer" to "base *base* integer" if the base is set to a number other than 10.

When the quit option is chosen (and allowed), q is returned along with the return code 3. The valrange module will not produce any output. It returns zero for success and non-zero for failure.

NAME

`ckstr` – display a prompt; verify and return a string answer

SYNOPSIS

`ckstr` [`-Q`] [`-W` *width*] [[`-r` *regexp*] [...]] [`-l` *length*] [`-d` *default*] [`-h` *help*]
[`-e` *error*]
[`-p` *prompt*] [`-k` *pid* [`-s` *signal*]]

`/usr/sadm/bin/errstr` [`-W`] [`-e` *error*]
`/usr/sadm/bin/helpstr` [`-W`] [`-h` *help*]
`/usr/sadm/bin/valstr` *input*

DESCRIPTION

`ckstr` prompts a user and validates the response. It defines, among other things, a prompt message whose response should be a string, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

The answer returned from this command must match the defined regular expression and be no longer than the length specified. If no regular expression is given, valid input must be a string with a length less than or equal to the length defined with no internal, leading or trailing white space. If no length is defined, the length is not checked. Either a regular expression or a length must be given with the command.

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The `-W` option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the `ckstr` command. They are `errstr` (which formats and displays an error message), `helpstr` (which formats and displays a help message), and `valstr` (which validates a response). These modules should be used in conjunction with FACE objects. In this instance, the FACE object defines the prompt.

The options and arguments for this command are:

`-Q` Specifies that quit will not be allowed as a valid response.

`-W` Specifies that prompt, help and error messages will be formatted to a line length of *width*.

`-r` Specifies a regular expression, `regexp`, against which the input should be validated. May include white space. If multiple expressions are defined, the answer must match only one of them.

`-l` Specifies the maximum length of the input.

`-d` Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

`-h` Defines the help messages as *help*.

`-e` Defines the error message as *error*.

`-p` Defines the prompt message as *prompt*.

`-k` Specifies that process ID *pid* is to be sent a signal if the user chooses to abort.

`-s` Specifies that the process ID *pid* defined with the `-k` option is to be sent signal `signal` when quit is chosen. If no signal is specified, `SIGTERM` is used.

input Input to be verified against format length and/or regular expression criteria.

EXIT CODES

0 = Successful execution
1 = EOF on input
2 = Usage error
3 = User termination (quit)

SEE ALSO

`valtools`(1).

NOTES

The default prompt for `ckstr` is:

```
Enter an appropriate value [?,q]:
```

The default error message is dependent upon the type of validation involved. The user will be told either that the length or the pattern matching failed.

The default help message is also dependent upon the type of validation involved. If a regular expression has been defined, the message is:

```
Please enter a string which matches the following pattern:
regexp
```

Other messages define the length requirement and the definition of a string.

When the quit option is chosen (and allowed), `q` is returned along with the return code `3`. The `valstr` module will not produce any output. It returns zero for success and non-zero for failure.

NAME

cktime – display a prompt; verify and return a time of day

SYNOPSIS

cktime [-Q] [-W *width*] [-f *format*] [-d *default*] [-h *help*] [-e *error*] [-p *prompt*] [-k *pid* [-s *signal*]]

/usr/sadm/bin/errtime [-W] [-e *error*] [-f *format*]
/usr/sadm/bin/helptime [-W] [-h *help*] [-f *format*]
/usr/sadm/bin/valtime [-f *format*] *input*

DESCRIPTION

cktime prompts a user and validates the response. It defines, among other things, a prompt message whose response should be a time, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return). The user response must match the defined format for the time of day.

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The -W option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the cktime command. They are errtime (which formats and displays an error message), helptime (which formats and displays a help message), and valtime (which validates a response). These modules should be used in conjunction with FML objects. In this instance, the FML object defines the prompt. When *format* is defined in the errtime and helptime modules, the messages will describe the expected format.

The options and arguments for this command are:

-Q Specifies that quit will not be allowed as a valid response.

-W Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-f Specifies the format against which the input will be verified. Possible formats and their definitions are:

%H = hour (00 - 23)
%I = hour (00 - 12)
%M = minute (00 - 59)
%p = ante meridian or post meridian
%r = time as %I:%M:%S %p
%R = time as %H:%M (the default format)
%S = seconds (00 - 59)
%T = time as %H:%M:%S

-d Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

-h Defines the help messages as *help*.

-e Defines the error message as *error*.

-p Defines the prompt message as *prompt*.

-k Specifies that process ID *pid* is to be sent a signal if the user chooses to abort.

-s Specifies that the process ID *pid* defined with the -k option is to be sent signal signal when quit is chosen. If no signal is specified, SIGTERM is used.

input Input to be verified against format criteria.

EXIT CODES

0 = Successful execution
1 = EOF on input
2 = Usage error
3 = User termination (quit)
4 = Garbled format argument

SEE ALSO

valtools(1).

NOTES

The default prompt for cktime is:

```
Enter the time of day [?,q]:
```

The default error message is:

```
ERROR - Please enter the time of day, using the following for-
mat:
```
format

The default help message is:

```
Please enter the time of day, using the following format:
```
format

When the quit option is chosen (and allowed), q is returned along with the return code 3. The valtime module will not produce any output. It returns zero for success and non-zero for failure.

NAME

ckuid – prompt for and validate a user ID

SYNOPSIS

ckuid [-Q] [-W *width*] [-m] [-d *default*] [-h *help*] [-e *error*] [-p *prompt*] [-k *pid* [-s *signal*]]

/usr/sadm/bin/erruid [-W] [-e *error*]
/usr/sadm/bin/helpuid [-W] [-m] [-h *help*]
/usr/sadm/bin/valuid *input*

DESCRIPTION

ckuid prompts a user and validates the response. It defines, among other things, a prompt message whose response should be an existing user ID, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The -W option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the ckuid command. They are erruid (which formats and displays an error message), helpuid (which formats and displays a help message), and valuid (which validates a response). These modules should be used in conjunction with FML objects. In this instance, the FML object defines the prompt.

The options and arguments for this command are:

-Q Specifies that quit will not be allowed as a valid response.

-W Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-m Displays a list of all logins when help is requested or when the user makes an error.

-d Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

-h Defines the help messages as *help*.

-e Defines the error message as *error*.

-p Defines the prompt message as *prompt*.

-k Specifies that process ID *pid* is to be sent a signal if the user chooses to abort.

-s Specifies that the process ID *pid* defined with the -k option is to be sent signal signal when quit is chosen. If no signal is specified, SIGTERM is used.

input Input to be verified against /etc/passwd.

EXIT CODES

0 = Successful execution
1 = EOF on input
2 = Usage error

3 = User termination (quit)

SEE ALSO

valtools(1).

NOTES

The default prompt for ckuid is:

```
Enter the login name of an existing user [?,q]:
```

The default error message is:

```
ERROR - Please enter the login name of an existing user.
Select the help option (?) for a list of valid login names.
```

. *(Last line appears only if the* -m *option of* ckuid *is used)*

The default help message is:

```
Please enter the login name of an existing user.
```

(If the -m *option of* ckuid *is used, a list of valid groups is also displayed.)*

When the quit option is chosen (and allowed), q is returned along with the return code 3. The valuid module will not produce any output. It returns zero for success and non-zero for failure.

NAME

ckyorn – prompt for and validate yes/no

SYNOPSIS

ckyorn [-Q] [-W *width*] [-d *default*] [-h *help*] [-e *error*] [-p *prompt*]
[-k *pid* [-s *signal*]]

/usr/sadm/bin/erryorn [-W] [-e *error*]
/usr/sadm/bin/helpyorn [-W] [-h *help*]
/usr/sadm/bin/valyorn *input*

DESCRIPTION

ckyorn prompts a user and validates the response. It defines, among other things, a prompt message for a yes or no answer, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The -W option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the ckyorn command. They are erryorn (which formats and displays an error message), helpyorn (which formats and displays a help message), and valyorn (which validates a response). These modules should be used in conjunction with FACE objects. In this instance, the FACE object defines the prompt. sp The options and arguments for this command are:

-Q Specifies that quit will not be allowed as a valid response.

-W Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-d Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

-h Defines the help messages as *help*.

-e Defines the error message as *error*.

-p Defines the prompt message as *prompt*.

-k Specifies that process ID *pid* is to be sent a signal if the user chooses to abort.

-s Specifies that the process ID *pid* defined with the -k option is to be sent signal signal when quit is chosen. If no signal is specified, SIGTERM is used.

input Input to be verified as y, yes, Y, Yes, YES or n, no, N, No, NO.

EXIT CODES

0 = Successful execution
1 = EOF on input
2 = Usage error
3 = User termination (quit)

SEE ALSO

valtools(1).

NOTES

The default prompt for ckyorn is:

```
Yes or No [y,n,?,q]:
```

The default error message is:

```
ERROR - Please enter yes or no.
```

The default help message is:

```
To respond in the affirmative, enter y, yes, Y, or YES.
To respond in the negative, enter n, no, N, or NO.
```

When the quit option is chosen (and allowed), q is returned along with the return code 3. The valyorn module will not produce any output. It returns zero for success and non-zero for failure.

NAME

co – check out RCS revisions

SYNOPSIS

co [*options*] *file* ...

DESCRIPTION

Co retrieves revisions from RCS files. Each file name ending in ',v' is taken to be an RCS file. All other files are assumed to be working files. Co retrieves a revision from each RCS file and stores it into the corresponding working file.

Pairs of RCS files and working files may be specified in 3 ways (see also the example section).

1) Both the RCS file and the working file are given. The RCS file name is of the form *path1/workfile*,v and the working file name is of the form *path2/workfile*, where *path1/* and *path2/* are (possibly different or empty) paths and *workfile* is a file name.

2) Only the RCS file is given. Then the working file is created in the current directory and its name is derived from the name of the RCS file by removing *path1/* and the suffix ',v'.

3) Only the working file is given. Then the name of the RCS file is derived from the name of the working file by removing *path2/* and appending the suffix ',v'.

If the RCS file is omitted or specified without a path, then co looks for the RCS file first in the directory ./RCS and then in the current directory.

Revisions of an RCS file may be checked out locked or unlocked. Locking a revision prevents overlapping updates. A revision checked out for reading or processing (e.g., compiling) need not be locked. A revision checked out for editing and later checkin must normally be locked. Locking a revision currently locked by another user fails. (A lock may be broken with the rcs(1) command.) Co with locking requires the caller to be on the access list of the RCS file, unless he is the owner of the file or the superuser, or the access list is empty. Co without locking is not subject to access list restrictions.

A revision is selected by number, checkin date/time, author, or state. If none of these options are specified, the latest revision on the trunk is retrieved. When the options are applied in combination, the latest revision that satisfies all of them is retrieved. The options for date/time, author, and state retrieve a revision on the *selected branch*. The selected branch is either derived from the revision number (if given), or is the highest branch on the trunk. A revision number may be attached to one of the options -l, -p, -q, or -r.

A co command applied to an RCS file with no revisions creates a zero-length file. Co always performs keyword substitution (see below).

-l[*rev*] locks the checked out revision for the caller. If omitted, the checked out revision is not locked. See option -r for handling of the revision number *rev*.

-p[*rev*] prints the retrieved revision on the std. output rather than storing it in the working file. This option is useful when co is part of a pipe.

-q[*rev*] quiet mode; diagnostics are not printed.

-d*date* retrieves the latest revision on the selected branch whose checkin date/time is less than or equal to *date*. The date and time may be given in free format and are converted to local time. Examples of formats for *date*:

```
22-April-1982, 17:20-CDT,
2:25 AM, Dec. 29, 1983,
Tue-PDT, 1981, 4pm Jul 21        (free format),
Fri, April 16 15:52:25 EST 1982  (output of ctime).
```

Most fields in the date and time may be defaulted. co determines the defaults in the order year, month, day, hour, minute, and second (most to least significant). At least one of these fields must be provided. For omitted fields that are of higher significance than the highest provided field, the current values are assumed. For all other omitted fields, the lowest possible values are assumed. For example, the date "20, 10:30" defaults to 10:30:00 of the 20th of the current month and current year. The date/time must be quoted if it contains spaces.

-r[*rev*] retrieves the latest revision whose number is less than or equal to *rev*. If *rev* indicates a branch rather than a revision, the latest revision on that branch is retrieved. *Rev* is composed of one or more numeric or symbolic fields separated by '.'. The numeric equivalent of a symbolic field is specified with the -n option of the commands ci and rcs.

-s*state* retrieves the latest revision on the selected branch whose state is set to *state*.

-w[*login*] retrieves the latest revision on the selected branch which was checked in by the user with login name *login*. If the argument *login* is omitted, the caller's login is assumed.

-j*joinlist* generates a new revision which is the join of the revisions on *joinlist*. *Joinlist* is a comma-separated list of pairs of the form *rev2:rev3*, where *rev2* and *rev3* are (symbolic or numeric) revision numbers. For the initial such pair, *rev1* denotes the revision selected by the options -l, ..., -w. For all other pairs, *rev1* denotes the revision generated by the previous pair. (Thus, the output of one join becomes the input to the next.)

For each pair, co joins revisions *rev1* and *rev3* with respect to *rev2*. This means that all changes that transform *rev2* into *rev1* are applied to a copy of *rev3*. This is particularly useful if *rev1* and *rev3* are the ends of two branches that have *rev2* as a common ancestor. If *rev1* < *rev2* < *rev3* on the same branch, joining generates a new revision which is like *rev3*, but with all changes that lead from *rev1* to *rev2* undone. If changes from *rev2* to *rev1* overlap with changes from *rev2* to *rev3*, co prints a warning and includes the overlapping sections, delimited by the lines
<<<<<<< *rev1*, =======, and >>>>>>> *rev3*.

For the initial pair, *rev2* may be omitted. The default is the common ancestor. If any of the arguments indicate branches, the latest revisions on those branches are assumed. If the option -l is present, the initial *rev1* is locked.

Keyword Substitution

Strings of the form *$keyword$* and *$keyword:...$* embedded in the text are replaced with strings of the form *$keyword: value $*, where *keyword* and *value* are pairs listed below. Keywords may be embedded in literal strings or comments to identify a revision.

Initially, the user enters strings of the form *$keyword$*. On checkout, co replaces these strings with strings of the form *$keyword: value $*. If a revision containing strings of the latter form is checked back in, the value fields will be replaced during the next checkout. Thus, the keyword values are automatically updated on checkout.

Keywords and their corresponding values:

$Author$ The login name of the user who checked in the revision. IMAGEN. Class$

$Date$ The date and time the revision was checked in.

$Header$ A standard header containing the RCS file name, the revision number, the date, the author, and the state.

$Locker$ The login name of the user who locked the revision (empty if not locked).

Log The log message supplied during checkin, preceded by a header containing the RCS file name, the revision number, the author, and the date. Existing log messages are NOT replaced. Instead, the new log message is inserted after `$Log:...$`. This is useful for accumulating a complete change log in a source file.

$Revision$ The revision number assigned to the revision.

$Source$ The full pathname of the RCS file.

$State$ The state assigned to the revision with `rcs -s` or `ci -s`.

$What$ The working file name and the revision number, preceded by the string `@(#)` recognized by `what`(1).

File Modes

The working file inherits the read and execute permissions from the RCS file. In addition, the owner write permission is turned on, unless the file is checked out unlocked and locking is set to `strict` (see `rcs`(1)).

If a file with the name of the working file exists already and has write permission, co aborts the checkout if `-q` is given, or asks whether to abort if `-q` is not given. If the existing working file is not writable, it is deleted before the checkout.

The caller of the command must have write permission in the working directory, read permission for the RCS file, and either read permission (for reading) or read/write permission (for locking) in the directory which contains the RCS file.

EXAMPLES

Suppose the current directory contains a subdirectory 'RCS' with an RCS file 'io.c,v'. Then all of the following commands retrieve the latest revision from 'RCS/io.c,v' and store it into 'io.c'.

```
co io.c;   co RCS/io.c,v;   co io.c,v;
co io.c RCS/io.c,v;   co io.c io.c,v;
co RCS/io.c,v io.c;   co io.c,v io.c;
```

FILES

A number of temporary files are created. A semaphore file is created in the directory of the RCS file to prevent simultaneous update.

DIAGNOSTICS

The RCS file name, the working file name, and the revision number retrieved are written to the diagnostic output. The exit status always refers to the last file checked out, and is 0 if the operation was successful, 1 otherwise.

SEE ALSO

`ci`(1), `ident`(1), `rcs`(1), `rcsdiff`(1), `rcsintro`(1), `rcsmerge`(1), `rlog`(1), `rcsfile`(4), `sccstorcs`(8).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering,* IEEE, Tokyo, Sept. 1982.

NOTES

The option `-d` gets confused in some circumstances, and accepts no date before 1970. There is no way to suppress the expansion of keywords, except by writing them differently. In `nroff` and `troff`, this is done by embedding the null-character '\&' into the keyword.

The option `-j` does not work for files that contain lines with a single '.'.

NAME

`cof2elf` – translate object file from COFF to ELF

SYNOPSIS

`cof2elf` [`-iqV`] [`-Q{yn}`] [`-s` *directory*] *files*

DESCRIPTION

`Cof2elf` converts one or more COFF object *files* to ELF. This translation occurs in place, meaning the original file contents are modified. If an input file is an archive, each member will be translated as necessary, and the archive will be rebuilt with its members in the original order. `Cof2elf` does not change input files that are not COFF.

Options have the following meanings:

`-i` Normally, the files are modified only when full translation occurs. Unrecognized data, such as unknown relocation types, are treated as errors and prevent translation. When `-i` is specified, `cof2elf` ignores these partial translation conditions and modifies the file anyway.

`-q` Normally, `cof2elf` prints a message for each file it examines, telling whether the file was translated, ignored, etc. The `-q` option (for `quiet`) suppresses these messages.

`-Q`*arg* If *arg* is `y`, identification information about `cof2elf` will be added to the output files; this can be useful for software administration. If *arg* is `n` (the default), this information is suppressed.

`-s`*directory*
By default, `cof2elf` modifies the input files. This option directs `cof2elf` to save a copy of the original files in the specified *directory*, which must exist. `Cof2elf` does not save files it does not modify.

`-V` This option directs `cof2elf` to print a version message on standard error.

SEE ALSO

`ld`(1), `elf`(3E), `a.out`(4), `ar`(4).

NOTES

Some debugging information is discarded. Although this does not affect the behavior of a running program, it may affect the information available for symbolic debugging.

`Cof2elf` translates only COFF relocatable files. It does not translate executable or static shared library files for two main reasons. First, the operating system supports executable files and static shared libraries, making translation unnecessary. Second, those files have specific address and alignment constraints determined by the file format. Matching the constraints with a different object file format is problematic.

When possible, programmers should recompile their source code to build new object files. `Cof2elf` is provided for those situations where source code is unavailable.

NAME

comb – combine SCCS deltas

SYNOPSIS

comb [–o] [–s] [–p*sid*] [–c*list*] *files*

DESCRIPTION

Comb generates a shell procedure (see sh(1)) that reconstructs the given SCCS files. The reconstructed files will usually be smaller than the original files. The arguments may be specified in any order, but all options 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 options are as follows. Each is explained as though only one named file is to be processed, but the effects of any option apply independently to each named file.

–p*SID* The SCCS *ID*entification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

–c*list* A *list* (see get(1) for the syntax of a *list*) of deltas to be preserved. All other deltas are discarded.

–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. Using the –o option 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 makes comb generate a shell procedure that produces 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
You should use this option before any SCCS files are actually combined, to determine how much space is saved by the combining process.

If you supply no options, 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.

DIAGNOSTICS

Use help(1) for explanations.

SEE ALSO

admin(1), delta(1), get(1), help(1), prs(1), sh(1), sccsfile(4).

NOTES

Comb may rearrange the shape of the tree of deltas. It may not save any space; in fact, the reconstructed file can be larger than the original.

NAME

cpp – the C language preprocessor

SYNOPSIS

/lib/cpp [*option* ...] [*ifile* [*ofile*]]

DESCRIPTION

cpp is the C language preprocessor. Thus, the output of cpp is designed to be in a form acceptable as input to the next pass of the C compiler. You should specify preprocessing by using the -E or -P option to cc(1), rather than by invoking /lib/cpp explicitly.

cpp optionally accepts two file names as arguments. *Ifile* and *ofile* are respectively the input and output for the preprocessor. They default to standard input and standard output if not supplied.

Options

–P Preprocess the input without producing the line control information used by the next pass of the C compiler.

–C By default, cpp strips C-style comments. If the –C option is specified, all comments (except those found on cpp directive lines) are passed along.

–U*name*
Remove any initial definition of *name*. *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. The –D option has lower precedence than the –U option. That is, if the same name is used in both a –U option and a –D option, the name will be undefined regardless of the order of the options.

–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 " " will be searched for first in the directory of the file with the #include line, 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 file with the #include line is not searched.

–T Forces cpp to use only the first eight characters for distinguishing different preprocessor names. This behavior is the same as for previous preprocessors with respect to the length of names and is included for backward compatability.

–Y*dir* Use directory *dir* in place of the standard list of directories when searching for #include files.

–H Print the path names of included files (one per line) on standard error.

Special Names

Two special names are understood by cpp. The name __LINE__ is defined as the current line number (as a decimal integer) as known by cpp, and __FILE__ is defined as the current file name (as a C string) as known by cpp. You can use them anywhere (including in macros) just as any other defined name.

Directives

All cpp directives start with #. Any number of blanks and tabs are allowed between the # and the directive. The directives are:

`#define` *name token-string*
: Replace subsequent instances of *name* with *token-string*.

`#define` *name*`(` *arg*`,` `...`, *arg* `)` *token-string*
: Replace subsequent instances of *name* followed by a `(`, a list of comma-separated set of tokens, and a `)` by *token-string*, where each occurrence of an *arg* in *token-string* is replaced by the corresponding set of tokens in the list. When a macro with arguments is expanded, the arguments are placed into the expanded *token-string* unchanged. After the entire *token-string* has been expanded, `cpp` restarts its scan for names to expand at the beginning of the newly created *token-string*.

  Notice that there can be no space between *name* and the `(`.

`#undef` *name*
: Forget the definition of *name* (if any).

`#ident`*string*
: Put *string* into the .comment section of an object file.

`#include` "*filename*"
`#include` <*filename*>
: Include at this point the contents of *filename* (which will then be run through `cpp`). When you use the <*filename*> notation, *filename* is only searched for in the standard places. See also the `-I` option above.

`#line` *integer-constant* "*filename*"
: Makes `cpp` 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 you omit *filename*, the current filename is unchanged.

`#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 will appear in the output if *name* has been the subject of a previous `#define` without being the subject of an intervening `#undef`.

`#ifndef` *name*
: The lines following will not appear in the output if *name* has been the subject of a previous `#define` without being the subject of an intervening `#undef`.

`#if` *constant-expression*
: Lines following will appear in the output if the *constant-expression* evaluates to non-zero. All binary non-assignment C operators, the `?:` operator, the unary `-`, `!`, and `~` operators are legal in *constant-expression*. The precedence of the operators is the same as defined by the C language.

  An unary operator is also `defined`, which can be used in *constant-expression* in these two forms: `defined(`*name*`)` or `defined` *name*. This lets you use `#ifdef` and `#ifndef` in a `#if` directive. In *constant-expression*, use only operators, integer constants, and names that `cpp` knows. The `sizeof` operator is not available.

`#elif` *constant-expression*
: Lines following will 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 in 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.

  To test whether or not either of two symbols, *bob* and *ted*, are defined, use

  #if defined(bob)|defined(ted)

`#else` Reverses the notion of the test directive that matches this directive. If lines previous to this directive are ignored, the following lines will appear in the output, and vice versa.

The test directives and the possible `#else` directives can be nested.

FILES

`/usr/include` Standard directory for `#include` files

DIAGNOSTICS

`cpp` error messages are intended to be self-explanatory. The line number and filename where the error occurred are printed along with the diagnostic.

SEE ALSO

cc(1).

NOTES

When new-line characters were found in argument lists for macros to be expanded, previous versions of `cpp` put out the new-lines as they were found and expanded. The current version of `cpp` replaces these new-lines with blanks.

NAME

cprs – compress a common object file

SYNOPSIS

cprs [-p] *file1 file2*

DESCRIPTION

Cprs reduces the size of a common object file, *file1*, by removing duplicate structure, enumeration, 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).

NAME

cscope – interactively examine a C program

SYNOPSIS

cscope [*options*] *files*...

DESCRIPTION

cscope is an interactive screen-oriented tool that allows the user to browse through C source files for specified elements of code.

By default, cscope examines the C (.c and .h), lex (.l), and yacc (.y) source files in the current directory. cscope may also be invoked for source files named on the command line. In either case, cscope searches the standard directories for #include files that it does not find in the current directory. cscope uses a symbol cross-reference, cscope.out by default, to locate functions, function calls, macros, variables, and preprocessor symbols in the files.

cscope builds the symbol cross-reference the first time it is used on the source files for the program being browsed. On a subsequent invocation, cscope rebuilds the cross-reference only if a source file has changed or the list of source files is different. When the cross-reference is rebuilt, the data for the unchanged files are copied from the old cross-reference, which makes rebuilding faster than the initial build.

The following options can appear in any combination:

–b	Build the cross-reference only.
–C	Ignore letter case when searching.
–c	Use only ASCII characters in the cross-reference file, that is, do not compress the data.
–d	Do not update the cross-reference.
–e	Suppress the ^e command prompt between files.
–f *reffile*	Use *reffile* as the cross-reference file name instead of the default cscope.out.
–I *incdir*	Look in *incdir* (before looking in *INCDIR*, the standard place for header files, normally /usr/include) for any #include files whose names do not begin with / and that are not specified on the command line or in *namefile* below. (The #include files may be specified with either double quotes or angle brackets.) The *incdir* directory is searched in addition to the current directory (which is searched first) and the standard list (which is searched last). If more than one occurrence of –I appears, the directories are searched in the order they appear on the command line.
–i *namefile*	Browse through all source files whose names are listed in *namefile* (file names separated by spaces, tabs, or new-lines) instead of the default (cscope.files). If this option is specified, cscope ignores any files appearing on the command line.
–L	Do a single search with line-oriented output when used with the *–num pattern* option.
–l	Line-oriented interface (see "Line-Oriented Interface" below).
–num pattern	Go to input field *num* (counting from 0) and find *pattern*.
–P *path*	Prepend *path* to relative file names in a pre-built cross-reference file so you do not have to change to the directory where the cross-

reference file was built. This option is only valid with the `-d` option.

`-p` *n* — Display the last *n* file path components instead of the default (1). Use 0 to not display the file name at all.

`-s` *dir* — Look in *dir* for additional source files. This option is ignored if source files are given on the command line.

`-T` — Use only the first eight characters to match against C symbols. A regular expression containing special characters other than a period (.) will not match any symbol if its minimum length is greater than eight characters.

`-U` — Do not check file time stamps (assume that no files have changed).

`-u` — Unconditionally build the cross-reference file (assume that all files have changed).

`-V` — Print on the first line of screen the version number of `cscope`.

The `-I`, `-p`, and `-T` options can also be in the `cscope.files` file.

Requesting the Initial Search

After the cross-reference is ready, `cscope` will display this menu:

```
Find this C symbol:
Find this function definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Press the TAB key repeatedly to move to the desired input field, type the text to search for, and then press the RETURN key.

Issuing Subsequent Requests

If the search is successful, any of these single-character commands can be used:

`1-9`	Edit the file referenced by the given line number.
`SPACE`	Display next set of matching lines.
`+`	Display next set of matching lines.
`–`	Display previous set of matching lines.
`^e`	Edit displayed files in order.
`>`	Append the displayed list of lines to a file.
`|`	Pipe all lines to a shell command.

At any time these single-character commands can also be used:

`TAB`	Move to next input field.
`RETURN`	Move to next input field.
`^n`	Move to next input field.
`^p`	Move to previous input field.
`^y`	Search with the last text typed.
`^b`	Move to previous input field and search pattern.
`^f`	Move to next input field and search pattern.
`^t`	Toggle ignore/use letter case when searching. (When ignoring letter case, search for `FILE` will match `File` and `file`.)

`^r`	Rebuild the cross-reference.
`!`	Start an interactive shell (type `^d` to return to `cscope`).
`^l`	Redraw the screen.
`?`	Give help information about `cscope` commands.
`^d`	Exit `cscope`.

Note: If the first character of the text to be searched for matches one of the above commands, escape it by typing a \ (backslash) first.

Substituting New Text for Old Text

After the text to be changed has been typed, `cscope` will prompt for the new text, and then it will display the lines containing the old text. Select the lines to be changed with these single-character commands:

`1-9`	Mark or unmark the line to be changed.
`*`	Mark or unmark all displayed lines to be changed.
`SPACE`	Display next set of lines.
`+`	Display next set of lines.
`-`	Display previous set of lines.
`a`	Mark all lines to be changed.
`^d`	Change the marked lines and exit.
`ESCAPE`	Exit without changing the marked lines.
`!`	Start an interactive shell (type `^d` to return to `cscope`).
`^l`	Redraw the screen.
`?`	Give help information about `cscope` commands.

Special Keys

If your terminal has arrow keys that work in `vi(1)`, you can use them to move around the input fields. The up-arrow key is useful to move to the previous input field instead of using the `TAB` key repeatedly. If you have the `CLEAR`, `NEXT`, or `PREV` keys they will act as the `^l`, `+`, and `-` commands, respectively.

Line-Oriented Interface

The `-l` option lets you use cscope where a screen-oriented interface would not be useful, e.g., from another screen-oriented program.

`cscope` will prompt with `>>` when it is ready for an input line starting with the field number (counting from 0) immediately followed by the search pattern, e.g., `1main` finds the definition of the `main` function.

If you just want a single search, instead of the `-l` option use the `-L` and *-num pattern* options, and you won't get the `>>` prompt.

For `-l`, `cscope` outputs the number of reference lines

```
cscope: 2 lines
```

For each reference found, `cscope` outputs a line consisting of the file name, function name, line number, and line text, separated by spaces, e.g.,

```
main.c main 161 main(argc, argv)
```

Note that the editor is not called to display a single reference, unlike the screen-oriented interface.

You can use the `r` command to rebuild the database.

`cscope` will quit when it detects end-of-file, or when the first character of an input line is `^d` or `q`.

ENVIRONMENT VARIABLES

`EDITOR`	Preferred editor, which defaults to `vi(1)`.

INCLUDEDIRS	Colon-separated list of directories to search for #include files.
HOME	Home directory, which is automatically set at login.
SHELL	Preferred shell, which defaults to sh(1).
SOURCEDIRS	Colon-separated list of directories to search for additional source files.
TERM	Terminal type, which must be a screen terminal.
TERMINFO	Terminal information directory full path name. If your terminal is not in the standard terminfo directory, see curses(3X) and terminfo(4) for how to make your own terminal description.
TMPDIR	Temporary file directory, which defaults to /var/tmp.
VIEWER	Preferred file display program [such as pg], which overrides EDITOR (see above).
VPATH	A colon-separated list of directories, each of which has the same directory structure below it. If VPATH is set, cscope searches for source files in the directories specified; if it is not set, cscope searches only in the current directory.

FILES

cscope.files	Default files containing -I, -p, and -T options and the list of source files (overridden by the -i option).
cscope.out	Symbol cross-reference file, which is put in the home directory if it cannot be created in the current directory.
ncscope.out	Temporary file containing new cross-reference before it replaces the old cross-reference.
INCDIR	Standard directory for #include files (usually /usr/include).

SEE ALSO

The "cscope" chapter in the *Programmer's Guide: ANSI C and Programming Support Tools*.

NOTES

cscope recognizes function definitions of the form:

fname blank (*args*) *white arg_decs white* {

where:

fname	is the function name
blank	is zero or more spaces or tabs, not including newlines
args	is any string that does not contain a " or a newline
white	is zero or more spaces, tabs, or newlines
arg_decs	are zero or more argument declarations (*arg_decs* may include comments and white space)

It is not necessary for a function declaration to start at the beginning of a line. The return type may precede the function name; cscope will still recognize the declaration. Function definitions that deviate from this form will not be recognized by cscope.

The Function column of the search output for the menu option Find functions called by this function: input field will only display the first function called in the line, that is, for this function

```
e()
{
     return (f() + g());
}
```

the display would be

```
Functions called by this function: e

File Function Line
```

```
a.c   f   3 return(f() + g());
```

Occasionally, a function definition or call may not be recognized because of braces inside `#if` statements. Similarly, the use of a variable may be incorrectly recognized as a definition.

A `typedef` name preceding a preprocessor statement will be incorrectly recognized as a global definition, e.g.,

```
LDFILE *
#if AR16WR
```

Preprocessor statements can also prevent the recognition of a global definition, e.g.,

```
char flag
#ifdef ALLOCATE_STORAGE
     = -1
#endif
;
```

A function declaration inside a function is incorrectly recognized as a function call, e.g.,

```
f()
{
     void g();
}
```

is incorrectly recognized as a call to `g()`.

cscope recognizes C++ classes by looking for the class keyword, but doesn't recognize that a `struct` is also a class, so it doesn't recognize inline member function definitions in a structure. It also doesn't expect the class keyword in a `typedef`, so it incorrectly recognizes `X` as a definition in

```
typedef class X * Y;
```

It also doesn't recognize operator function definitions

```
Bool Feature::operator==(const Feature & other)
{
     ...
}
```

NAME

`ctags` – create a tags file

SYNOPSIS

`ctags [ -BFatuwvx ]` *name* ...

DESCRIPTION

`Ctags` makes a tags file for `ex(1)` from the specified C, Pascal and Fortran sources. A tags file gives the locations of specified objects (in this case functions and typedefs) in a group of files. Each line of the tags file contains the object name, the file in which it is defined, and an address specification for the object definition. Functions are searched for with a pattern, typedefs with a line number. Specifiers are given in separate fields on the line, separated by blanks or tabs. Using the `tags` file, `ex` can quickly find these object definitions.

If the `-x` flag is given, `ctags` produces a list of object names, the line number and file name on which each is defined, as well as the text of that line and prints this information on the standard output. This is a simple index which can be printed out.

If the `-v` flag is given, an index of a different form is produced on the standard output. This listing contains the function name, file name, and page number (assuming 64 line pages).

Files whose names end in or are assumed to be C source files and are searched for C routine and macro definitions. Others are first examined to see if they contain any Pascal or Fortran routine definitions; if not, they are processed again looking for C definitions.

Other options are:

`-a` append to tags file.

`-w` suppressing warning diagnostics.

`-u` causing the specified files to be *updated* in tags, that is, all references to them are deleted, and the new values are appended to the file. (Beware: this option is implemented in a way which is rather slow; it is usually faster to simply rebuild the `tags` file.)

`-F` use forward searching patterns (/.../) (default).

`-B` use backward searching patterns (?...?).

`-t` create tags for typedefs.

The tag *main* is treated specially in C programs. The tag formed is created by prepending *M* to the name of the file, with a trailing .c removed, if any, and leading pathname components also removed. This makes use of `ctags` practical in directories with more than one program.

FILES

tags output tags file

SEE ALSO

`ex(1)`, `vi(1)`.

BUGS

Recognition of `functions`, `subroutines` and `procedures` for FORTRAN and Pascal is done in a very simpleminded way. No attempt is made to deal with block structure; if you have two Pascal procedures in different blocks with the same name, the procedure will not work.

Does not know about #ifdefs.

Should know about Pascal types. Relies on the input being well formed to detect typedefs. Use of −tx shows only the last line of typedefs.

NAME

`ctl` – COFF-to-legend translator

SYNOPSIS

`ctl` [*option*] *filename*

DESCRIPTION

The `ctl` command translates the debugging information stored inside an object module from COFF format to legend format. Normally, `ctl` is invoked automatically by the compiler (via an `as`(1) option); consult the man page for your compiler to see if it does this.

`Ctl` accepts options both on the command line, and from the LEGENDS environment variable. In cases of conflicting options, command line options override LEGENDS options, then option precedence is from right to left (with right-most options having the highest precedence).

Many important ctl options are described by the `legend`(5) manual page. In addition, the following options are interpreted by `ctl`:

`-fix-bb`
: Indicate that the compiler generates a redundant pair of begin-block and end-block symbols around each function. This option should be used with `gcc`.

`-h"`[*string*]`"`
: Store the given string in the legend. This switch is generally used to indicate which compiler was used.

`-l`[*language*]
: Specify which source language was used; possible values are `fortran`, `c`, `ansi-c`, and `pascal`. The default is `c`.

`-ocs`
: Assume an 88k-OCS-compliant frame format. If this switch is omitted, then it is assumed that r30 is the frame pointer. This switch is ignored if a `.coffsem` or `sem`[`.`*value*] symbol is present in the object module.

`-reverse-arrays`
: Indicate that array dimensions are stored in reverse of the source code order. This switch is ignored if a `.coffsem` or `sem`[`.`*value*] symbol is present in the object module.

`-s"`[*pathname*]`"`
: Indicate that *pathname* is the source file for the object module being translated.

FILES

`file.o` object file
`file.lg` optional debugging information file

SEE ALSO

`as` (1), `cc`(1), `gcc`(1), `ghcc`(1), `ghf77`(1), `ghpc`(1), `mxdb`(1), `legend`(5).

NAME

ctrace − trace a C program to debug it

SYNOPSIS

ctrace [*options*] [*file*]

DESCRIPTION

ctrace lets you 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 omit *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 is 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.

Commonly used *options* 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. The *options* that print variables in additional formats are:

−o Octal
−x Hexadecimal
−u Unsigned
−e Floating point

Other *options* for special circumstances are:

−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 using 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 below 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 cc(1) preprocessor options.

The *options* that tailor the run-time trace package for the traced program to run in a non-UNIX system environment are:

−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).

−Q*arg*
: If *arg* is `y`, identification information about `ctrace` will be added to the output files. This can be useful for software administration. Giving `n` for *arg* exlicitly asks for no such information, which is the default behavior.

−V
: Prints version information on the standard error.

EXAMPLES

If the file `lc.c` contains the following 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 the following commands and test data:

```
cc lc.c
a.out
1
(ctrl-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 (ctrl-d), the final output will be:

```
         /* c == -1 */
10       printf("%d\n", nl);
         /* nl == 2 */2
          return
```

Note 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 comment 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 '\n' in line 8. Once your attention is drawn to the `if` statement in line 8, you will probably realize that you used the assignment operator (=) in place of the equal 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. The default 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
```

These functions can also be called from `sdb(1)` if they are compiled 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 the static variable `tr_ct_` to 0 and 1, respectively.

FILES

`runtime.c` run-time trace package

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.

SEE ALSO

`signal`(2), `ctype`(3C), `fflush`(3S), `longjmp`(3C), `printf`(3S), `setjmp`(3C), `string`(3C).

NOTES

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.

Pointer values are always treated as pointers to character strings.

`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.

The loop trace output elimination is done separately for each file of a multi-file program. Separate output elimination 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.

NAME

cxref – generate C program cross-reference

SYNOPSIS

cxref [*options*] *files*

DESCRIPTION

Cxref analyzes a collection of C files and builds a cross-reference table. Cxref uses a special version of cc to include #define'd information in its symbol table. It generates a list of all symbols (auto, static, and global) in each individual file, or, with the -c option, in combination. The table includes four fields: NAME, FILE, FUNCTION, and LINE. The line numbers appearing in the LINE field also show reference marks as appropriate. The reference marks include:

assignment	=
declaration	–
definition	*

If no reference marks appear, you can assume a general reference.

Options

Cxref interprets the -D, -I, -U options in the same manner that cc does. In addition, cxref interprets the following options:

-c Combine the source files into a single report. Without the -c option, cxref generates a separate report for each file on the command line.

-o *file* Direct output to *file*.

-s Operates silently; does not print input file names.

-t Format listing for 80-column width.

-w*num* Width option that formats output no wider than *num* (decimal) columns. This option will default to 80 if *num* is not specified or is less than 51. These options are accepted only in an ELF environment:

-d Disables printing declarations, making the report easier to read.

-l Does not print local variables. Prints only global and file scope statistics.

-C Runs only the first pass of cxref, creating a .cx file that can later be passed to cxref. This is similar to the -c option of cc or lint.

-F Prints the full path of the referenced file names.

-L*cols* Modifies the number of columns in the LINE field. If you do not specify a number, cxref defaults to five columns.

-V Prints version information on the standard error.

-W*name*, *file*, *function*, *line*
Set the width of each field (*name*, *file*, *function*, and *line* are non-negative integers). The default widths are:

Field	Characters
NAME	15
FILE	13
FUNCTION	15
LINE	20 (4 per column)

EXAMPLE

a.c

```
1       main()
2       {
3               int i;
4               extern char c;
5
6               i=65;
7               c=(char)i;
8       }
```

Resulting cross-reference table:

```
NAME   FILE         FUNCTION   LINE
c      a.c          ---        4-      7=
i      a.c          main       3*      6=     7
main   a.c          ---        2*
u3b2   predefined   ---        0*
unix   predefined   ---        0*
```

FILES

TMPDIR/tcx.*	temporary files
TMPDIR/cx.*	temporary files
LIBDIR/xref	accessed by cxref
LIBDIR	usually /usr/lib
TMPDIR	usually /usr/tmp but can be redefined by setting the environment variable TMPDIR [see tempnam in tmpnam(3S)].

DIAGNOSTICS

Error messages usually mean you cannot compile the files.

SEE ALSO

cc(1), lint(1).

NAME

dbx – source level debugger

SYNOPSIS

dbx [-r] [-s *style*] [-i] [-I *dir*] [*objfile* [*corefile*]]

DESCRIPTION

The dbx utility is a tool for source-level debugging and execution of programs under the DG/UX system. *Objfile* is an executable file–one that has been compiled and linked. The compiler must use the appropriate flag(s) to produce symbol information in the object file. The machine-level facilities of dbx can be used on any program not linked with the -s option.

If no *objfile* is specified, dbx looks for a file named a.out in the current directory.

When a *corefile* is specified, dbx can be used to examine the state of the program when it faulted.

If the file .dbxinit exists in the current directory, dbx executes the debugger commands in it. Dbx also checks for .dbxinit in the user's home directory if there is not one in the current directory.

Options are:

-r Execute *objfile* immediately. The object filename must be supplied. Parameters follow the object filename. When the program terminates, the reason for termination is reported and the user can enter the debugger or let the program fault. Dbx reads from /dev/tty when -r is specified and standard input is not a terminal.

-s *style* Inform dbx of the style of the symbol names in the executable. By convention, *style* is the compile command that produced the executable, e.g. cc, gcc, ghcc, or ghf77. The -s option is required only when debugging a COFF executable whose of debugging information differs in form from that produced by cc, the default style.

-i Force dbx to act as though standard input is a terminal.

-I *dir* Add *dir* to the list of directories that dbx searches when looking for a source file. Normally dbx looks for source files in the current directory and in the directory where *objfile* is located. The directory search path can also be set with the use command.

Unless -r is specified, dbx just prompts and waits for a command.

Expressions and Scope

Dbx evaluates an expression according to the scope that is in effect at the time the expression is evaluated. This scope determines which variables are accessible. For example, the command

```
stop at "foo.c":5 if a == 17
```

contains the expression "a == 17", which will be evaluated when line 5 of the file foo.c is reached. At that time, the variable *a* must be either a local variable of the current function or a global variable. The expression "a == 17" must be a legal C language expression.

Execution and Tracing Commands

`run` [*args*] [< *filename*] [>|>> *filename*]
: Execute the *objfile* specified on the `dbx` command line or the one specified with the most recent debug command. *Args* are passed as command line arguments. Input and output can be redirected using the symbols <, >, and >>. Other characters in *args* are passed through unchanged. If no arguments are specified, the argument list from the last `run` command (if any) is used. If *objfile* has been written since the last time the symbolic information was read in, `dbx` reads the new information before beginning execution.

`rerun` [*args*] [< *filename*] [>|>> *filename*]
: Except in the case where no arguments are specified, `rerun` is identical to `run`. When no arguments are specified, `rerun` runs the program with no arguments at all.

`debug` *objfile* [*corefile*]
: Stop debugging the current program (if any), and begin debugging the program found in *objfile* with the given *corefile*. This process avoids the overhead required to reinitialize `dbx`.

`kill` Stop debugging the current process, kill the process, but leave `dbx` ready to debug another.

`trace` *source-line-number* [`if` *condition*]
`trace` @*label*[*offset*] [`if` *condition*]
`trace` *procedure/function*[*offset*] [`if` *condition*]
`trace` *expression* `at` *source-line-number* [`if` *condition*]
`trace` *variable* [`in` *procedure/function*] [`if` *condition*]
: Print tracing information when the program is executed. A number is associated with the `trace` command, which may later be used to turn the tracing off (see the `delete` and `status` commands).

The first argument describes what is to be traced. If it specifies a source statement (by line number, label, or offset from a procedure or function), the line or label is printed immediately before being executed. An offset is + or − some number of lines.

If the argument is a simple procedure or function name, every time it is called information is printed telling what routine called it, from what source line it was called, and what parameters were passed to it.

If the argument is an expression with an `at` clause, the value of the expression is printed whenever the identified source line is reached.

If the argument is a variable, the name and value of the variable are printed whenever it changes. The clause `in` *procedure/function* restricts tracing information to be printed only while executing inside the specified procedure or function.

Source line numbers and function names may be qualified by a filename and following colon, as in `"mumble.c":17` (quotes are optional).

Condition is a boolean expression and is evaluated before printing the tracing information; if it is false, the information is not printed.

`stop at` *source-line-number* [`if` *condition*]
`stop` *@label*[*offset*] [`if` *condition*]
`stop in` *procedure/function*[*offset*] [`if` *condition*]
`stop` *variable* [`if` *condition*]

Stop execution when the given line or label is reached, the procedure or function is called, or the variable is changed.

`status` [`>` *filename*]

Print out the currently active `trace` and `stop` commands.

`commands` [*command-number*]

Attach a series of commands to the specified `trace` or `stop` command (or to the last one that was set) to be performed whenever the `trace` or `stop` is taken. The commands, which may be any debugger commands including those that resume or redirect execution, are entered on successive lines and delimited by the `end` command on a separate line. You may use an `if/then/else` construct to specify alternate actions based on a conditional expression.

`delete` *command-number* [,*command-number...*]

Remove the traces or stops corresponding to the given numbers. The numbers associated with traces and stops are printed by the `status` command. `Delete all` removes all traces and stops.

`clear` [*source-line-number*]
`clear` *@label*
`clear` *procedure/function*
`clear` *variable*

Delete all traces or stops set on the given line-number, label, function, or variable. `Clear` without argument clears all traces or stops on the line at which execution is stopped.

`catch` [*signal* [,*signal...*]]
`ignore` [*signal* [,*signal...*]]

Start or stop trapping the specified signals before they are sent to the program; a `signal` may be identified by its number or its name. This command is useful when a program being debugged handles signals such as interrupts. Initially, all signals are trapped except SIGCONT, SIGCLD, SIGALRM, and SIGKILL. Without arguments, `catch` and `ignore` display a list of signals currently trapped or ignored.

`cont` [*n*]

Continue execution. If *n* is specified, ignore the current breakpoint until it has been reached this number of times. Execution cannot be continued if the process has called the standard procedure 'exit'. `Dbx` tries to keep the process from exiting, thereby letting the user examine the program state.

`position` *source-line-number*
`position` *procedure/function*[*offset*]
`position` *@label*[*offset*]

Set the current instruction pointer to the indicated position. Execution does not resume until directed by the user. Positioning to a different stack frame may have unpredictable results.

`jump` *source-line-number*
`jump` *procedure/function*[*offset*]
`jump` @*label*[*offset*]
Continue execution from the given source line, procedure, or label.

`finish`
Continue execution until the current frame is exited.

`step` [*n*]
Execute one or more source lines.

`next` [*n*]
Execute one or more source lines, but do not follow procedure or function calls. The difference between `next` and `step` is that if a line contains a call to a procedure or function, `step` stops at the beginning of that block, whereas `next` continues execution to the next immediate source line.

Displaying and Naming Data

`print`[/*format*] *expression* [, *expression* ...]
Print out the values of the expressions. The optional *format* is one of `x` (hexadecimal), `d` (signed decimal), `u` (unsigned decimal), `o` (octal), `c` (character), or `b` (binary). A valid expression may refer to variables in the current procedure; it may also invoke any procedure or function in the program.

`call` *subroutine* [(*arg* [,*arg*...])]
Call a FORTRAN 77 subroutine.

`whatis` *name*
Print the declaration of the given name. In debugging COFF executables, longs are reported as ints, and tags are reported as typedefs.

`assign` *variable* = *expression*
`set` *variable* = *expression*
Assign the value of the expression to the variable.

`where` [*n*]
Display the call/return stack. If *n* is specified and *n* < 0, show the bottommost *n* frames of the stack. If *n* is specified and *n* > 0, show the topmost *n* frames of the stack.

`up` [*n*] Move up the call stack *n* levels in the direction of main. If *n* is not specified, the default is 1. This command allows you to examine the local variables in functions other than the current one.

`down` [*n*]
Move down the call stack *n* levels towards the current stopping point. If *n* is not specified, the default is 1.

`describe` [*procedure/function*]
Describe the current or specified procedure or function, including its name, address, and source coordinates.

`describe` *source-line-number*
`describe` @*label*
Describe the given source line or label, including the associated starting address and the name of the program block.

`args` Display the arguments to the current procedure or function.

`dump` [> *filename*]
Print the names and values of all local variables.

`echo` *string*
Print a constant string; C escape sequences must be used to print newlines and leading or trailing whitespace.

Accessing Source Files

`edit` [*filename*]
`edit` *procedure/function-name*
Invoke an editor on *filename* or the current source file if none is specified. If a *procedure* or *function* name is specified, the editor is invoked on the file that contains it. The default editor depends on the installation. To override the default, set the environment variable EDITOR to the name of the desired editor.

`file` [*filename*]
Change the current source filename to *filename*. If you omit *filename*, the current source filename is printed.

`func` [*procedure/function*]
Change the current function. If no function is specified, print the name of current function. Changing the current function implicitly changes the current source file to the one that contains the function.

`list` [[*filename*:]*linespec* [, *linespec*]]
List the lines in the current (or specified) source file from the first line specified through the second, or print a window of lines surrounding a single line. If no lines are specified, list 10 more lines. A *linespec* may be a source line number, label, or function name with optional offset. It may also be a simple offset (+ or − some number), which specifies an offset from the last line printed, or from the first of two *linespecs* in a `list` command. `$` used as a line number means the last line in the file.

`pwd` Print `dbx`'s notion of the working directory.

`cd` *directory*
Change `dbx`'s working directory. The change does not take effect for the program being debugged until the next time it is started.

`use` *directory-list*
Set the list of directories to be searched when `dbx` looks for source files.

Machine-level Commands

address *,address/* [*mode*]
[*address*] / [*n*] [*mode*]
Print the contents of memory starting at the first *address* and continuing up to the second *address* or until *n* items are printed. If no address is specified, the address following the one printed most recently is used. *Mode* specifies how memory is to be printed; if *mode* is omitted, the previous mode specified is used. The initial mode is `H`. The following modes are supported:

`i`	a machine instruction
`d`	a short word in decimal
`D`	a long word in decimal
`o`	a short word in octal
`O`	a long word in octal
`x`	a short word in hexadecimal
`X`	a long word in hexadecimal
`b`	two bytes in octal
`c`	two bytes as characters
`s`	a string of characters terminated by a null byte
`f`	a single precision real number
`g`	a double precision real number

Symbolic addresses are specified by preceding the name with an &. Registers are referred to with the following symbolic names:

`$r0`	zero
`$r1`	subroutine return pointer
`$r2-$r9`	called procedure parameter registers
`$r10-$r13`	called procedure temporary registers
`$r14-$r25`	calling procedure reserved registers
`$r26-$r29`	linker
`$r30`	frame pointer
`$r31`	stack pointer
`$fp`	frame pointer (register 30)
`$sp`	stack pointer (register 31)
`$fpsr`	floating-point status register
`$fpcr`	floating-point control register
`$psr`	processor status register
`$sxip`	shadow execute instruction pointer
`$snip`	shadow next instruction pointer
`$sfip`	shadow fetched instruction pointer
`$cfa`	canonical frame address pseudo-register
`$pc`	program counter pseudo-register

Addresses may be expressions made up of other addresses and the operators +, −, and indirection (unary *).

`stepi` [*n*]
`nexti` [*n*]
Single step as in `step` or `next`, but do a single instruction rather than source line.

`tracei` [*address*] [`if` *condition*]
`tracei` [*variable*] [`at` *address*] [`if` *condition*]
`stopi` [`at`] [*address*] [`if` *condition*]
Turn on tracing or set a stop using a machine instruction address.

`position` *address*
Set the current instruction pointer to the specified address.

Miscellaneous Commands

sh [*command-line*]
: Pass the command line to the shell for execution. Without argument, sh suspends the debugging session and enters a shell. The SHELL environment variable determines which shell is used.

define *macro-name*
: Define a macro with the given name; the body of the macro is entered on successive lines and delimited by the end command on a separate line. Arguments to the macro are denoted by #1, #2, and so on.

alias [*new-command-name* [*character-sequence*]]
: Respond to *new-command-name* as though it were *character-sequence*. Arguments to the alias are permitted, and are denoted by #1, #2, and so on. Invoked with *new-command-name* only, alias prints the *character-sequence* associated with *new-command-name*. Invoked without arguments, alias prints a list of currently defined aliases.

save *filename*
: Save the state of the debugging session in the specified file (if file exists, it is first deleted). The state comprises stop and trace commands (with any associated commands), user-defined macros, and aliases.

restore *filename*
: Restore the debugger state saved in the specified file.

help [*command*]
: Print out a summary of dbx commands, or a synopsis of the given command.

source *filename*
: Read dbx commands from the given *filename*. Especially useful when the *filename* has been created by redirecting a status command from an earlier debugging session.

style *stylename*
: Inform dbx of the style of the symbol names in the executable to be debugged. By convention, *stylename* is the compile command originally used to produce the executable: currently valid *stylenames* are cc, gcc, ghcc, and ghf77. The default style is cc.

: When debugging ELF executables, the style command serves no purpose, and is ignored.

quit
: Exit from dbx.

FILES

a.out	Object file
.dbxinit	Initial commands

SEE ALSO

cc(1), gcc(1), ghcc(1), ghf77(1).

NOTES

Non-local goto commands can cause some trace/stops to be missed.

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 permanently introduces into the named SCCS file changes that were made to the file retrieved by get(1) (called the *g-file*, or generated file).

Delta adds a change 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 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 on options specified and flags (see admin(1)) that may be present in the SCCS file (see -m and -y options below).

Options apply independently to each named file.

-r*SID* Uniquely identifies which delta is to be made to the SCCS file. This option is necessary only if two or more outstanding gets for editing (get -e) on the same SCCS file were done by the same person (login name). The SID value specified with the -r option 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 deltas SID, as well as the number of lines inserted, deleted and unchanged in the SCCS file.

-n Retains the edited *g-file* (normally removed at completion of delta processing).

-g*list* Specifies a *list* (see get(1) for the definition of *list*) of deltas 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 option).

MRs in a list are separated by blanks and/or tab characters. An unescaped newline 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) that will validate the MR numbers. If a non-zero exit status is returned from the MR number validation program, delta terminates (assumes that the MR numbers were not all valid).

-y[*comment*] Arbitrary text that describes 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 Prints (on the standard output) the SCCS file differences before and after the delta is applied in a `diff`(1) format.

EXAMPLES

```
delta /work/archives/s.file1
```

This command permanently installs any changes done to 'file1' (the g-file), which must be in the current working directory, into the SCCS file 's.file1' in the directory `/work/archives`.

```
delta -ytest -n -p s.file2
```

This command permanently installs any changes done to 'file2' (the g-file) into the SCCS file 's.file2', including adding the description found in 'test' as the reason for making the change, as well as not removing the file 'file2' from the current directory. The -p will list the before and after differences of the SCCS file.

FILES

All files of the form ?-file are explained in *Programmer's Guide: ANSI C and Programming Support Tools* The naming convention for these files is also described there.

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*.
/bin/bdiff	Program to compute differences between the "gotten" file and the *g-file*.

DIAGNOSTICS

Use `help`(1) for explanations.

SEE ALSO

`admin`(1), `bdiff`(1), `cdc`(1), `comb`(1), `get`(1), `help`(1), `prs`(1), `rmdel`(1). `sccsfile`(4) in the *Programmer's Reference for the DG/UX System (Volume 2)* "Source Code Control System" in *Programmer's Guide: ANSI C and Programming Support Tools.*

NOTES

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, use multiple `get/delta` sequences.

If the standard input (–) is specified on the *delta* command line, the –m (if necessary) and –y options *must* also be present. Omission of these options causes an error to occur.

Comments are limited to text strings of at most 512 characters.

NAME

dis – object code disassembler

SYNOPSIS

dis [-o] [-V] [-d *sec*] [-D *sec*] [-F *function*] [-t *sec*] [-l *string*] *file* ...

DESCRIPTION

Dis produces an assembly language listing of *file*, which may be an object file or, in an ELF environment, an archive of object files. The listing includes assembly statements and a hexadecimal representation of the binary that produced those statements.

In an ELF environment, dis accepts the following options, which may be specified in any order.

-d *sec*	Disassemble the named section as data, printing the offset of the data from the beginning of the section.
-D *sec*	Disassemble the named section as data, printing the actual address of the data.
-F *function*	Disassemble only the named function in each object file specified on the command line. The -F option may be specified multiple times on the command line.
-l *string*	Disassemble the archive file specified by *string*. For example, one would issue the command dis -l x -l z to disassemble libx.a and libz.a, which are assumed to be in *LIBDIR*.
-o	Print numbers in octal. The default is hexadecimal.
-t *sec*	Disassemble the named section as text.
-V	Print, on standard error, the version number of the disassembler being executed.

If the -d, -D or -t options are specified, only those named sections from each user-supplied file name will be disassembled. Otherwise, all sections containing text will be disassembled.

On output, a number enclosed in brackets at the beginning of a line, such as [5], indicates that the break-pointable line number starts with the following instruction. These line numbers will be printed only if the file was compiled with additional debugging information [e.g., the -g option of cc]. An expression such as <40> in the operand field or in the symbolic disassembly, following a relative displacement for control transfer instructions, is the computed address within the section to which control will be transferred. A function name will appear in the first column, followed by ( ) if the object file contains a symbol table.

FILES

LIBDIR usually /usr/lib

DIAGNOSTICS

The self-explanatory diagnostics indicate errors in the command line or problems encountered with the specified files.

SEE ALSO

as(1), cc(1), ld(1), a.out(4).

NOTES

At this time, symbolic disassembly does not take advantage of additional information available if the file is compiled with the -g option.

NAME

`fsplit` – split f77 or ratfor files

SYNOPSIS

`fsplit` *options files*

DESCRIPTION

`Fsplit` splits the named *file(s)* into separate files, with one procedure per file. A procedure includes *blockdata*, *function*, *main*, *program*, and *subroutine* program segments. Procedure *X* is put in file *X*`.f`, or *X*`.r` depending on the language option chosen, with the following exceptions: *main* is put in the file *MAIN*`.[fr]` and unnamed *blockdata* segments in the files *blockdataN*`.[fr]` where *N* is a unique integer value for each file.

The following *options* pertain:

`-f` (default) Input files are *f77*.

`-r` Input files are `ratfor`.

`-s` Strip `f77` input lines to 72 or fewer characters with trailing blanks removed.

SEE ALSO

`csplit`(1), `f77`(1), `ratfor`(1), `split`(1).

NAME

gcc – GNU C language compiler

SYNOPSIS

gcc [*option*] ... *file* ...

DESCRIPTION

The GNU C compiler uses a command syntax much like the Unix C compiler. The gcc program accepts options and file names as operands. Multiple single-letter options may *not* be grouped: '-dr' is very different from '-d -r'. When you invoke gcc, it normally does preprocessing, compilation, assembly and linking. File names that end in .c are taken as C source to be preprocessed and compiled; compiler output files plus any input files with names ending in .s are assembled; then the resulting object files, plus any other input files, are linked together to produce an executable. Command options allow you to stop this process at an intermediate stage. For example, the -c option says not to run the link editor. Then the output consists of object files output by the assembler. Other command options are passed on to one stage. Some options control the preprocessor and others the compiler itself.

Some options are accepted only by one or the other version of GNU C. Such options are indicated below by "(V1)" or "(V2)".

OPTIONS

Here are the options to control the overall compilation process, including those that say whether to link, whether to assemble, and so on.

-V *version*
: The argument *version* specifies which version of GNU C to run. This is useful when multiple versions are installed. For example, *version* might be 2, meaning to run GNU C version 2.

-c
: Compile or assemble the source files, but do not link. Produce object files with names made by replacing .c or .s with .o at the end of the input file names. Do nothing at all for object files specified as input.

-S
: Compile into assembler code but do not assemble. The assembler output file name is made by replacing .c with .s at the end of the input file name. Do nothing at all for assembler source files or object files specified as input.

-E
: Run only the C preprocessor. Preprocess all the C source files specified and output the results to standard output.

-o *file*
: Place output in file *file*. This applies to any output being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code. If -o is not specified, the default is to put an executable file in a.out, the object file *source*.c in *source*.o, an assembler file in *source*.s, and preprocessed C on standard output.

-v
: Compiler driver program prints the commands it executes as it runs the preprocessor, compiler proper, assembler and link editor. Some of these are directed to print their own version numbers.

-pipe
: Run preprocessor, compiler, and assembler in parallel, connected via pipelines. You should not use this option if your system does not have enough physical

memory to support all four processes simultaneously.

Options Controlling Language

These options determine the dialect of C that the compiler accepts:

`-ansi`

Support all ANSI standard C programs. This turns off certain features of GNU C that are incompatible with ANSI C, such as the `asm`, `inline` and `typeof` keywords, and predefined macros such as `unix` that identify the type of system you are using. It also enables the rarely-used ANSI trigraph feature.

The `-ansi` option does not cause non-ANSI programs to be rejected gratuitously. For that, `-pedantic` is required in addition to `-ansi`. The macro `__STRICT_ANSI__` is predefined when the `-ansi` option is used. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros that the ANSI standard doesn't call for; this is to avoid interfering with any programs that might use these names for other things.

`-fno-asm`

Do not recognize `asm`, `inline` or `typeof` as a keyword. These words may then be used as identifiers. `-ansi` implies `-fno-asm`.

`-trigraphs`

Support ANSI C trigraphs. The `-ansi` option also has this effect.

`-traditional`

Attempt to support some aspects of traditional C compilers. Specifically:

* All `extern` declarations take effect globally even if they are written inside a function definition. This includes implicit declarations of functions.
* The keywords `typeof`, `inline`, `signed`, `const` and `volatile` are not recognized.
* Comparisons between pointers and integers are always allowed.
* Integer types `unsigned short` and `unsigned char` promote to `unsigned int`.
* Out-of-range floating point literals are not an error.
* All automatic variables not declared `register` are preserved by `longjmp`. Ordinarily, GNU C follows ANSI C: automatic variables not declared `volatile` may be clobbered.
* In the preprocessor, comments convert to nothing at all, rather than to a space. This allows traditional token concatenation.
* In the preprocessor, macro arguments are recognized within string constants in a macro definition (and their values are stringified, though without additional quote marks, when they appear in such a context). The preprocessor also considers a string constant to end at a newline.
* The predefined macro `__STDC__` is not defined when you use `-traditional`, but `__GNUC__` is (since the GNU extensions which `__GNUC__` indicates are not affected by `-traditional`). If you need to write header files that work differently depending on whether `-traditional` is in use, by testing both of these predefined macros you can distinguish four situations: GNU C, traditional GNU C, other ANSI C compilers, and other old C compilers.

* String literals are put into the writable data section instead of into read-only text.

`-fcond-mismatch`
Allow conditional expressions with mismatched types in the second and third arguments. The value of such an expression is void.

`-funsigned-char`
Let the type `char` be unsigned, like `unsigned char`. The type `char` is always a distinct type from either `signed char` or `unsigned char`, even though its behavior is always just like one of those two.

`-fsigned-char`
Let the type `char` be signed, like `signed char`.

`-fwritable-strings`
Store string constants in the writable data segment and represent identical strings distinctly (don't share storage). This is for compatibility with old programs which assume they can write into string constants.

Options to Request or Suppress Warnings

`-w` Inhibit all warning messages.

`-pedantic`
Issue all the warnings demanded by strict ANSI standard C; reject all programs that use forbidden extensions. Valid ANSI standard C programs should compile properly with or without this option (though a rare few will require `-ansi`). However, without this option, certain GNU extensions and traditional C features are supported as well. With this option, they are rejected.

`-pedantic-errors` (V2)
Like `-pedantic`, except that errors are produced rather than warnings. This option is supported only in Version 2 of GNU C.

`-W` Print extra warning messages for these events:

* An automatic variable is used without first being initialized. These warnings are possible only in optimizing compilation, because they require data flow information that is computed only when optimizing. They occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared `volatile`, or whose address is taken, or whose size is other than 1, 2, 4, or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers. Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by the flow analysis pass before the warnings are printed. These warnings are made optional because GNU C is not smart enough to see all the reasons why the code might be correct despite appearing to have an error.
* A nonvolatile automatic variable might be changed by a call to `longjmp`. These warnings as well are possible only in optimizing compilation. The compiler sees only the calls to `setjmp`. It cannot know where `longjmp` will be called; in fact, a signal handler could call it at any point in the code. As a result, you may get a warning even when there is in fact no problem because `longjmp` cannot in fact be called at the place which would cause a problem.
* A function can return either with or without a value. (Falling off the end of the function body is considered returning without a value.) Spurious warnings can occur because GNU C does not realize that certain functions (including `abort` and `longjmp`) will never return.

* An expression-statement contains no side effects.

`-Wimplicit`
Warn whenever a function is implicitly declared.

`-Wreturn-type`
Warn whenever a function is defined with a return-type that defaults to `int`. Also warn about any `return` statement with no return-value in a function whose return-type is not `void`.

`-Wunused`
Warn whenever a local variable is unused aside from its declaration, whenever a function is declared static but never defined, and whenever a statement computes a result that is explicitly not used.

`-Wswitch`
Warn whenever a switch statement has an index of enumeral type and lacks a case for one or more of the named codes of that enumeration. (The presence of a default label prevents this warning.) Outside the enumeration range, case labels also provoke warnings when this option is used.

`-Wcomment`
Warn whenever a comment-start sequence '/*' appears in a comment.

`-Wtrigraphs`
Warn if any trigraphs are encountered (assuming they are enabled).

`-Wformat` (V2)
Check calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified.

`-Wall`
All of the above `-W` options combined.

`-Wtraditional` (V2)
Warn about certain constructs that behave differently in traditional and ANSI C.

`-Wshadow`
Warn whenever a local variable shadows another local variable.

`-Wid-clash-`*len*
Warn whenever two distinct identifiers match in the first *len* characters.

`-Wpointer-arith`
Warn about anything that depends on the "size of" a function type or of void. GNU C assigns these types a size of 1, for convenience in calculations with `void *` pointers and pointers to functions.

`-Wcast-qual`
Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a `const char *` is cast to an ordinary `char *`.

`-Wcast-align` (V2)
Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a `char *` is cast to an `int *` on machines where integers can only be accessed at two- or four-byte boundaries.

`-Wwrite-strings`
Give string constants the type `const char[`*length*`]` so that copying the address of one into a non-const `char *` pointer will get a warning. These warnings will help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using `const` in declarations and

prototypes. Otherwise, it will just be a nuisance; this is why `-Wall` does not request these warnings.

`-Wconversion` (V2)
Warn if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or signedness of a fixed point argument except when the same as the default promotion.

`-mwarn-passed-structs`
Emit a warning message if a structure is passed to a function, or declared as a function argument. This warns about the places where `gcc` will not interoperate with compilers that do not pass structures according to the *88open Object Compatibility Standard*.

Options for Debugging Your Program

`-g` Produce debugging information for `mxdb`, `dbx`, or `sdb`.

Unlike most other C compilers, GNU C allows you to use `-g` with `-O`. The shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops. Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

In the ELF environment, debugging information is in `legend`(5) format for all supported debuggers. An optional `LEGENDS` environment variable can contain special generation options such as "`-external`" to reduce link-time by storing most debugging information in a separate file. See `legend`(5) for details.

In a COFF environment, GNU C generates debugging information in `legend` format for use by `mxdb` when the `LEGENDS` environment variable is present; the information is in standard COFF format by default.

These three options, which control legend generation, are superseded by the use of the `LEGENDS` environment variable, and will be eliminated in the future:

`-mlegend`
Causes the assembler to invoke `ctl`(1), the COFF-to-legend translator.

`-mexternal-legend`
Causes the assembler to pass the `-external` option to `ctl`(1).

`-mkeep-coff`
Causes the assembler to pass the `-keep-std` option to `ctl`(1).

`-mocs-debug-info`
Put out additional debug information to comply with the *88open Object Compatibility Standard* text description information. This is the default.

`-mno-ocs-debug-info`
Do not put out any additional debugging information.

`-mocs-frame-position`
When emitting debugging information for automatic variables and parameters stored on the stack, use the offset from the canonical frame address (CFA), which is the stack pointer (register 31) when the function is entered. The CFA is

specified by the *88open Object Compatibility Standard*. This is the default behavior of GNU C.

`-mno-ocs-frame-position`
: When emitting debugging information for automatic variables and parameters stored on the stack, use the offset from the frame pointer register (register 30). When this option is in effect, the frame pointer is not eliminated when debugging information is selected by the `-g` switch.

`-p` Generate extra code to write profile information suitable for the analysis program `prof`.

Options Controlling Optimization

`-O` Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

Without `-O`, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code. Without `-O`, only variables declared `register` are allocated in registers.

With `-O`, the compiler tries to reduce code size and execution time. Some of the `-f` options described below turn specific kinds of optimization on or off.

`-O2` (V2)
: Highly optimize. All supported optimizations are performed. As compared to `-O`, this option will increase both compilation time and the performance of the generated code.

Options of the form `-f`*flag* specify machine-independent flags. Most flags have both positive and negative forms, as in `ffoo` and `fno-foo`. Only one of the forms is listed here: the one which is not the default.

`-ffloat-store`
: Do not store floating-point variables in registers.

`-fno-defer-pop`
: Always pop the arguments to each function call as soon as that function returns. Normally the compiler (when optimizing) lets arguments accumulate on the stack for several function calls and pops them all at once.

`-fforce-mem`
: Force memory operands to be copied into registers before doing arithmetic on them. This may produce better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load.

`-fforce-addr`
: Force memory address constants to be copied into registers before doing arithmetic on them. This may produce better code just as `-fforce-mem` may.

`-fomit-frame-pointer`
: Don't keep the frame pointer in a register for functions that don't need one. This eliminates the instructions that save, set up and restore frame pointers; it also makes an extra register available in many functions.

On an AViiON computer, if you specify `-O` and do not specify `-fno-omit-frame-pointer`, this is enabled automatically.

`-finline` (V2)
: Pay attention the `inline` keyword. Normally the negation of this option `-fno-inline` is used to keep the compiler from expanding any functions inline. However, the opposite effect may be desirable when compiling with `-g`, since `-g` normally turns off all inline function expansion.

`-finline-functions`
: Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. If all calls to a given function are integrated, and the function is declared `static`, then the function is normally not output as assembler code in its own right.

`-fcaller-saves`
: Enable values to be allocated in registers that will be clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls.

`-fkeep-inline-functions`
: Even if all calls to a given function are integrated, and the function is declared `static`, nevertheless output a separate run-time callable version of the function.

`-fno-function-cse`
: Do not put function addresses in registers; make each instruction that calls a constant function contain the function's address explicitly. This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.

These options control specific optimizations. All are implied by the `-O2` option.

`-fstrength-reduce`
: Perform the optimizations of loop strength reduction and elimination of iteration variables.

`-fthread-jumps` (V2)
: Perform optimizations where we check to see if a jump branches to a location where another comparison subsumed by the first is found. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.

`-funroll-loops` (V2)
: Perform the optimization of loop unrolling. This is only done for loops whose number of iterations can be determined at compile time or run time.

`-funroll-all-loops` (V2)
: Perform the optimization of loop unrolling. This is done for all loops. This usually makes programs run more slowly.

`-fcse-follow-jumps` (V2)
: In common subexpression elimination, scan through jump instructions in certain cases. This is not as powerful as completely global CSE, but not as slow either.

`-frerun-cse-after-loop` (V2)
: Re-run common subexpression elimination after loop optimizations has been performed.

`-fexpensive-optimizations` (V2)
: Perform a number of minor optimizations that are relatively expensive.

`-fdelayed-branch`
Reorder instructions to take advantage of the delay slot following branch and subroutine call instructions.

`-fschedule-insns` (V2)
If supported for the target machine, attempt to reorder instructions to eliminate execution stalls due to required data being unavailable.

`-fschedule-insns2` (V2)
Similar to `-fschedule-insns`, but requests an additional pass of instruction scheduling after register allocation has been done.

`-fcombine-regs` (V1)
Allow the combine pass to combine an instruction that copies one register into another. This might or might not produce better code when used in addition to `-O`.

Options Controlling the Preprocessor

These options control the C preprocessor, which is run on each C source file before actual compilation. If you use the `-E` option, nothing is done except C preprocessing. Some of these options make sense only together with `-E` because they request preprocessor output that is not suitable for actual compilation.

`-i` *file* (V2)
Process *file* as input, discarding the resulting output, before processing the regular input file. Because the output generated from *file* is discarded, the only effect of `-i` *file* is to make the macros defined in *file* available for use in the main input.

`-nostdinc`
Do not search the standard system directories for header files. Only the directories you have specified with `-I` options (and the current directory, if appropriate) are searched. Between `-nostdinc` and `-I-`, you can eliminate all directories from the search path except those you specify.

`-E` Run only the C preprocessor. Preprocess all the C source files specified and output the results to standard output.

`-C` Tell the preprocessor not to discard comments. Used with the `-E` option.

`-P` (V2)
Tell the preprocessor not to generate `#line` commands. Used with the `-E` option.

`-M` Tell the preprocessor to output a rule suitable for `make` describing the dependencies of each object file. For each source file, the preprocessor outputs one `make`-rule whose target is the object file name for that source file and whose dependencies are all the files `#included` in it. This rule may be a single line or may be continued with '\'-newline if it is long. `-M` implies `-E`.

`-MM`
Like `-M`, but the output mentions only the user-header files included with '`#include "`*file*`"`.' System header files included with '`#include <`*file*`>`' are omitted. `-MM` implies `-E`.

`-MD` (V2)
Like `-M` but the dependency information is written to files with names made by replacing `.c` with `.d` at the end of the input file names. This is in addition to compiling the file as specified: `-MD` does not inhibit ordinary compilation the way `-M` does.

-MMD (V2)
Like -MD but mention only user header files, not system header files.

-H Tell the preprocessor to output the names of include files to the standard error file, in addition to the normal processing.

-D*macro*
Define macro *macro* with the string '1' as its definition.

-D*macro*=*defn*
Define macro *macro* as *defn*.

-U*macro*
Undefine macro *macro*.

-trigraphs
Support ANSI C trigraphs. The -ansi option also has this effect.

Options for Linking

-l*library*
Search a standard list of directories for a library named *library*, which is actually a file named lib*library*.a. The link editor uses this file as if it had been specified precisely by name. The directories searched include several standard system directories plus any that you specify with -L. Normally the files found this way are library files--archive files whose members are object files. The link editor handles an archive file by scanning through it for members which define symbols that have so far been referenced but not defined. But if the file that is found is an ordinary object file, it is linked in the usual fashion. The only difference between an -l option and specifying a file name is that -l searches several directories.

-nostdlib
Don't use the standard system libraries and startup files when linking. Only the files you specify will be passed to the link editor.

-static
Produce a static object, that is an object which contains no shared objects. This option causes -dn to be added to the link line; see ld(1).

-shared
Produce a shared object. This option causes -G to be added to the link line, to produce a shared object which can then be linked with other objects to form an executable.

-symbolic
Bind references to global symbols when building a shared object. Warn about any unresolved references (unless overridden by the link editor option -z defs: see ld(1)). This option causes -Bsymbolic -G to be added to the link line.

Gcc also passes the options -e, -h, -n, -r, -s, -t, -u, -x, and -z to the link editor; see ld(1) for these options.

Options for Directory Search

-I*dir*
Search directory *dir* for include files.

-I-
Any directories specified with -I options before the -I- option are searched only for the case of '#include "*file*"'; they are not searched for '#include <*file*>'. If additional directories are specified with -I options after the -I-,

these directories are searched for all `#include` directives. (Ordinarily *all* `-I` directories are used this way.) In addition, the `-I-` option inhibits the use of the current directory as the first search directory for '`#include "`*file*`"`'. Therefore, the current directory is searched only if it is requested explicitly with '`-I.`'. Specifying both '`-I-`' and '`-I.`' allows you to control precisely which directories are searched before the current one and which are searched after.

`-L`*dir*
Add directory *dir* to the list of directories to be searched for `-l`.

`-B`*prefix*
Compiler driver program tries *prefix* as a prefix for each program it tries to run. These programs are `cpp`, `cc1`, `as` and `ld`. For each subprogram to be run, the compiler driver first tries the `-B` prefix, if any. If that name is not found, or if `-B` was not specified, the driver tries the standard prefix, which is `/usr/lib/gcc/gcc-`. If this does not result in a file name that is found, the unmodified program name is searched for, using the directories specified in your `PATH` environment variable.

The run-time support file `gnulib` is also searched for, using the `-B` prefix, if needed. If it is not found there, the standard prefix above is tried, and that is all. The file is left out of the link if it is not found by those means.

You can get a similar result from the environment variable `GCC_EXEC_PREFIX`. If it is defined, its value is used as a prefix in the same way. If both the `-B` option and the `GCC_EXEC_PREFIX` variable are present, the `-B` option is used first and the environment variable value second.

Options for Code Generation Conventions

`-fpic`
Generate position-independent code, suitable for use in a shared object.

`-mbig-pic`
Produce position-independent code that will work correctly if the global offset table of a shared object exceeds 16k. (Modules should be recompiled with this option when the link editor reports the error "Relocation overflows at *address*..." when producing a shared object.)

`-fpcc-struct-return`
Use the same convention for returning `struct` and `union` values that is used by PCC. This convention is less efficient for small structures, and on many machines it fails to be reentrant; but it has the advantage of allowing intercallability between GCC-compiled code and PCC-compiled code.

`-fshort-enums` (V2)
Allocate to an `enum` type only as many bytes as it needs for the declared range of possible values. Specifically, the `enum` type will be equivalent to the smallest integer type which has enough room.

`-fshared-data`
Requests that the data and non-`const` variables of this compilation be shared data rather than private data.

`-fno-common` (V2)
Allocate even unitialized global variables in the bss section of the object file, rather than generating them as common blocks. This has the effect that if the same variable is declared (without `extern`) in two different compilations, you

will get an error when you link them. The only reason this might be useful is if you wish to verify that the program will work on other systems which always work this way.

`-fvolatile`
Consider all memory references through pointers to be volatile.

`-fvolatile-global` (V1)
Consider all memory references to extern and global data items to be volatile.

`-ffixed-`*reg*
Treat the register named *reg* as a fixed register; generated code should never refer to it (except perhaps as a stack pointer, frame pointer or in some other fixed role). *reg* is one of `r0-r31`. Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results. This flag does not have a negative form, because it specifies a three-way choice.

`-fcall-used-`*reg*
Treat the register named *reg* as an allocatable register that is clobbered by function calls. It may be allocated for temporaries or variables that do not live across a call. Functions compiled this way will not save and restore the register *reg*. Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results. This flag does not have a negative form, because it specifies a three-way choice.

`-fcall-saved-`*reg*
Treat the register named *reg* as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way will save and restore the register *reg* if they use it. Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results. A different sort of disaster will result from the use of this flag for a register in which function values may be returned. This flag does not have a negative form, because it specifies a three-way choice.

`-mno-underscores`
Do not emit a leading underscore before all external names. This switch is useful for embedded systems and does not allow interoperation with the standard library.

`-mtrap-large-shift`
Emit a `tbnd` instruction before each shift by a non-constant amount, to trap if the shift count is less than zero or greater than 31. The 88000 produces unusual results in such cases, and the trap will halt the program at the point an out of range shift is done, rather than producing unexpected results. The ANSI standard for C specifies that shifts outside of the range of 0 to number_bits - 1 is undefined. It is an error to specify both `-mtrap-large-shift` and `-mhandle-large-shift`.

`-mhandle-large-shift`
Emit a four instruction sequence for each shift by a non-constant amount, if the shift count is less than zero or greater than 31. Logical shifts and arithmetic shifts left produce a 0 result if the shift count is out of bounds. Arithmetic shifts right produce a copy of the sign bit if the shift count is out of bounds. The ANSI standard for C specifies that shifts outside of the range of 0 to number_bits - 1 is undefined. It is an error to specify both `-mtrap-large-shift` and

-mhandle-large-shift.

-mno-check-zero-division
Do not emit code to check both the divisor and dividend when doing normal integer division (as opposed to unsigned division) to see if either is negative, and fixup things up so that the division is done with positive numbers. You would use this switch when you are confident that most or all signed divisions are done with positive numbers.

-muse-div-instruction
Do not emit code to check if an integer division by zero occurs and issue trap number 503 if it occurs.

If this fixup is not done, the 88100 will trap to the kernel if either number is negative. The operating system will calculate the correct answer for all negative operands, except for the most negative number (−2147833648) divided by negative 1, whose signed result cannot be represented in 32 bits.

-midentify-revision
Emit an assembly ident directive which gives the filename, date, time, and compiler revision, for use with the what command.

There are several macros you can define to control your source and target environments when developing applications. These macros control header files, function declarations, binary formats, and other aspects of the source and target environments. The macros are helpful when you are porting applications to or from non-DG/UX systems such as BSD or AT&T systems. The macros can also make development of POSIX- or BCS-conformant applications easier. For developing BCS-conformant applications, the SDE utility is also helpful. See *Porting Applications to the DG/UX™ System* and the sde-target(1), sdetab(4), and sde(5) manual pages.

FILES

file.c	input file
file.o	object file
a.out	loaded output
*TMPDIR/cc**	temporary files. *TMPDIR* is usually /usr/tmp but can be redefined by setting the environment variable TMPDIR.
/usr/lib/gcc/gcc-cpp	preprocessor
/usr/lib/gcc/gcc-cc1	compiler
/usr/lib/gcc/gcc-gnulib	library needed by gcc
/lib/crt0.o	runtime startup routine
/lib/libc.a	standard library, see intro(3)
/usr/include	standard directory for #include files

SEE ALSO

cc(1), as(1), ld(1), sde-target(1), sdetab(4), sde(5).

COPYING

one.
Permission is granted to copy and distribute translations of the gcc(1) manual page into another language, under the above conditions for modified versions, except that this permission notice may be included in translations approved by the Free Software Foundation instead of in the original English.

NAME

get – check out 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* ...

where:

SID	The SCCS identification string of a version of an SCCS file
cutoff	Date and time, in the form *YY*[*MM*[*DD*[*HH*[*MM*[*SS*]]]]]
list	A list of deltas in the following syntax: *list* ::= *range* | *list*,*range* *range* ::= *SID* | *SID*–*SID*
string	A string (must be quoted if it contains a space)
seq-no	The delta sequence number of the SCCS file delta (version) to be retrieved
file	Name of the file to be checked out

DESCRIPTION

Get generates an ASCII text file from each named SCCS file according to the specifications given by its options, which begin with –. The arguments may be specified in any order, but they all apply to all named SCCS files. If a directory is named, get treats each file in the directory 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* . Its name is derived from the SCCS filename by simply removing the leading s.; (see also *FILES*, below).

Each of the options is explained below as though only one SCCS file is to be processed, but the effects of any option applies independently to each named file.

–r*SID* Specify 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 option is also used), as a function of the SID specified.

–c*cutoff* Specify cutoff date and time. 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 two-digit pieces of the *cutoff* date-time. This feature lets you 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

–e Indicate 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 option 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 option is accidentally ruined in the process of editing it, it may be regenerated by re-executing the get command with the `-k` option in place of the `-e` option.

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` option is used.

`-b` Used with the `-e` option, indicate that the new delta should have an SID in a new branch as shown in Table 1. This option 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* has no successors on the SCCS file tree.) Note: A branch *delta* may always be created from a non-leaf *delta*.

`-i`*list* Specify a list of deltas to be included (forced to be applied) in the creation of the generated file. SID, the SCCS Identification of a delta, may be in any form shown in the "SID Specified" column of Table 1. Partial SIDs are interpreted as shown in the "SID Retrieved" column of Table 1.

`-x`*list* Specify a list of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the `-i` option for the *list* format.

`-k` Suppress replacement of identification keywords (see below) in the retrieved text by their value. The `-k` option is implied by the `-e` option.

`-l[p]` Write a delta summary 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` Write the text retrieved from the SCCS file to on the standard output. No *g-file* is created. All output that normally goes to the standard output goes to file descriptor 2 instead, unless the `-s` option is used. In that case, it disappears.

`-s` Suppress all output normally written on the standard output. However, fatal error messages (which always go to file descriptor 2) remain unaffected.

`-m` Precede each text line retrieved from the SCCS file 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` Precede each generated text line 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` options are used, the format is: %M% value, followed by a horizontal tab, followed by the `-m` option generated format.

`-g` Suppress 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` Access the most recently created (top) 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 `get`ing the file.

`-a`*seq-no* Specify the delta sequence number of the SCCS file delta (version) to be retrieved (see *sccsfile*(5)). This option is used by the `comb(1)` command; it is not a generally useful option, and users should not use it. If both the `-r` and `-a` options are specified, the `-a` option is used. Care should be taken when using the `-a` option in conjunction with the `-e` option, as the SID of the delta to be created may not be what one expects. The `-r` option can be used with the `-a` and `-e` options 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` option 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 filename is printed (preceded by a new-line) before it is processed. If the `-i` option is used, included deltas are listed following the notation "Included"; if the `-x` option is used, excluded deltas are listed following the notation "Excluded".

TABLE 1. Determination of SCCS Identification String

SID* Specified	-b Option 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` option 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 filename.
`%P%`	Fully qualified SCCS filename.
`%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 four-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%

EXAMPLES

```
get -e /work/archives/s.file1
```

This command generates an ASCII text file named 'file1' in the current working directory from the SCCS file 's.file1' in the directory `/work/archives`, while giving the new file proper attributes for editing or changing (delta). This also creates a file named 'p.file1' in the directory `/work/archives`.

FILES

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 filename is formed from the SCCS filename: the last component of all SCCS filenames 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 filenames 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` option is used). A *g-file* is created in all cases, whether or not any lines of text were generated by the `get`. It is owned by the real user. If the `-k` option 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` option 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 blank character if the delta was applied; * otherwise.
- A blank character if the delta was applied or was not applied and ignored; * if the delta was not applied and was not ignored.
- A code indicating a special reason why the delta was or was not applied:

  I: Included.
  X: Excluded.
  C: Cut off (by a `-c` option).
- Blank.
- SCCS identification (SID).
- Tab character.
- Date and time (in the form YY/MM/DD HH:MM:SS) of creation.
- Blank.
- 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` option along to *delta*. Its contents are also used to prevent a subsequent execution of `get` with an `-e` option 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` option if it was present, followed by a blank and the `-x` option 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 in mode 444.

DIAGNOSTICS

Use `help`(1) for explanations.

SEE ALSO

`admin`(1), `comb`(1), `delta`(1), `help`(1), `prs`(1), `unget`(1), `what`(1), `sccsfile`(4).

"Source Code Control System" in *Programmer's Guide: ANSI C and Programming Support Tools*.

NOTES

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 option is used.

NAME

`ident` – identify files

SYNOPSIS

`ident` *file* ...

DESCRIPTION

`Ident` searches the named files for all occurrences of the pattern `$`*keyword:...*`$`, where *keyword* is one of

Author
Date
Header
Locker
Log
Revision
Source
State
What

These patterns are normally inserted automatically by the RCS command `co(1)`, but can also be inserted manually.

`Ident` works on text files as well as object files. For example, if the C program in file f.c contains

```
char rcsid[] = "$Header:  Header information $";
```

and `f.c` is compiled into `f.o`, then the command

```
ident  f.c  f.o
```

will print

```
f.c:
     $Header:  Header information $
f.o:
     $Header:  Header information $
```

SEE ALSO

ci(1), **co**(1), **rcs**(1), **rcsdiff**(1), **rcsintro**(1), **rcsmerge**(1), **rlog**(1), **rcsfile**(4).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering,* IEEE, Tokyo, Sept. 1982.

NAME

ipcrm – remove a message queue, semaphore set, or shared memory ID

SYNOPSIS

ipcrm [*options*]

DESCRIPTION

ipcrm removes one or more messages, semaphores, or shared memory identifiers. The identifiers are specified by the following *options*:

–q *msqid* Remove the message queue identifier *msqid* from the system and destroy the message queue and data structure associated with it.

–m *shmid* Remove the shared memory identifier *shmid* from the system. The shared memory segment and data structure associated with it are destroyed after the last detach.

–s *semid* Remove the semaphore identifier *semid* from the system and destroy the set of semaphores and data structure associated with it.

–Q *msgkey* Remove the message queue identifier, created with key *msgkey*, from the system and destroy the message queue and data structure associated with it.

–M *shmkey* Removes the shared memory identifier, created with key *shmkey*, from the system. The shared memory segment and data structure associated with it are destroyed after the last detach.

–S *semkey* Remove the semaphore identifier, created with key *semkey*, from the system and destroy the set of semaphores and data structure associated with it.

The details of the removes are described in msgctl(2), shmctl(2), and semctl(2). Use the ipcs command to find the identifiers and keys.

SEE ALSO

ipcs(1), dg_sys_info(2), msgctl(2), msgget(2), semctl(2), semget(2), semop(2), shmctl(2), shmget(2), shmsys(2).

NOTE

The ipcs(1) command returns hex values for the message queue key, semaphore key, and shared memory key. If you use ipcs to return values for use by ipcrm(1) with the –S, –Q, or –M options, you must convert the values to decimal before giving them to ipcrm.

NAME

ipcs – report inter-process communication facilities status

SYNOPSIS

ipcs [*options*]

DESCRIPTION

ipcs prints information about active inter-process communication facilities. Without *options*, information is printed in short format for message queues, shared memory, and semaphores that are currently active in the system. Otherwise, the information that is displayed is controlled by the following *options*:

–q Print information about active message queues.

–m Print information about active shared memory segments.

–s Print information about active semaphores.

If –q, –m, or –s are specified, information about only those indicated is printed. If none of these three are specified, information about all three is printed subject to these options:

–b Print information on biggest allowable size: maximum number of bytes in messages on queue for message queues, size of segments for shared memory, and number of semaphores in each set for semaphores. See below for meaning of columns in a listing.

–c Print creator's login name and group name. See below.

–o Print information on outstanding usage: number of messages on queue and total number of bytes in messages on queue for message queues and number of processes attached to shared memory segments.

–p Print process number information: process ID of last process to send a message, process ID of last process to receive a message on message queues, process ID of creating process, and process ID of last process to attach or detach on shared memory segments. See below.

–t Print time information: time of the last control operation that changed the access permissions for all facilities, time of last msgsnd and last msgrcv on message queues, time of last shmat and last shmdt on shared memory, time of last semop on semaphores. See below.

–a Use all print options. (This is a shorthand notation for –b, –c, –o, –p, and –t.)

The column headings and the meaning of the columns in an ipcs listing are given below; the letters in parentheses indicate the options that cause the corresponding heading to appear; "all" means that the heading always appears. Note that these options only determine what information is provided for each facility; they do not determine which facilities are listed.

T (all) Type of the facility:
- q message queue
- m shared memory segment
- s semaphore

ID (all) The identifier for the facility entry.

KEY (all) The key used as an argument to msgget, semget, or shmget to create the facility entry. (Note: The key of a shared memory segment is changed to IPC_PRIVATE when the segment has been removed until all processes attached to the segment detach it.)

MODE	(all)	The facility access modes and flags: The mode consists of 11 characters that are interpreted as follows. The first two characters are: R A process is waiting on a *msgrcv*. S A process is waiting on a *msgsnd*. D The associated shared memory segment has been removed. It will disappear when the last process attached to the segment detaches it. C The associated shared memory segment is to be cleared when the first attach is executed. - The corresponding special flag is not set. The next nine characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions; the next to permissions of others in the user-group of the facility entry; and the last to all others. Within each set, the first character indicates permission to read, the second character indicates permission to write or alter the facility entry, and the last character is currently unused. The permissions are indicated as follows: r Read permission is granted. w Write permission is granted. a Alter permission is granted. - The indicated permission is not granted.
OWNER	(all)	The login name of the owner of the facility entry.
GROUP	(all)	The group name of the group of the owner of the facility entry.
CREATOR	(a,c)	The login name of the creator of the facility entry.
CGROUP	(a,c)	The group name of the group of the creator of the facility entry.
CBYTES	(a,o)	The number of bytes in messages currently outstanding on the associated message queue.
QNUM	(a,o)	The number of messages currently outstanding on the associated message queue.
QBYTES	(a,b)	The maximum number of bytes allowed in messages outstanding on the associated message queue.
LSPID	(a,p)	The process ID of the last process to send a message to the associated queue.
LRPID	(a,p)	The process ID of the last process to receive a message from the associated queue.
STIME	(a,t)	The time the last message was sent to the associated queue.
RTIME	(a,t)	The time the last message was received from the associated queue.
CTIME	(a,t)	The time when the associated entry was created or changed.
NATTCH	(a,o)	The number of processes attached to the associated shared memory segment.
SEGSZ	(a,b)	The size of the associated shared memory segment.
CPID	(a,p)	The process ID of the creator of the shared memory entry.

LPID	(a,p)	The process ID of the last process to attach or detach the shared memory segment.
ATIME	(a,t)	The time the last attach was completed to the associated shared memory segment.
DTIME	(a,t)	The time the last detach was completed on the associated shared memory segment.
NSEMS	(a,b)	The number of semaphores in the set associated with the semaphore entry.
OTIME	(a,t)	The time the last semaphore operation was completed on the set associated with the semaphore entry.

FILES

/dgux	system image (*namelist*)
/etc/passwd	user names
/etc/group	group names

SEE ALSO

dg_sys_info(2), semop(2), shmsys(2).

NOTES

Things can change while ipcs is running; the information it gives is guaranteed to be accurate only when it was retrieved.

NAME

`ld` – link editor for object files

SYNOPSIS

`ld` [*options*] *files* ...

DESCRIPTION

The `ld` command combines relocatable object files, performs relocation, and resolves external symbols. `ld` operates in two modes, static or dynamic, as governed by the `-d` option. In static mode, `-dn`, relocatable object files given as arguments are combined to produce an executable object file; if the `-r` option is specified, relocatable object files are combined to produce one relocatable object file. In dynamic mode, `-dy`, the default, relocatable object files given as arguments are combined to produce an executable object file that will be linked at execution with any shared object files given as arguments; if the `-G` option is specified, relocatable object files are combined to produce a shared object. In all cases, the output of `ld` is left in `a.out` by default.

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. The library may be either a relocatable archive or a shared object. For an archive library, only those routines defining an unresolved external reference are loaded. The archive library symbol table [see `ar`(4)] is searched sequentially with as many passes as are necessary to resolve external references that can be satisfied by library members. Thus, the ordering of members in the library is functionally unimportant, unless there exist multiple library members defining the same external symbol. A shared object consists of a single entity all of whose references must be resolved within the executable being built or within other shared objects with which it is linked.

The following options are recognized by `ld`:

`-a` In static mode only, produce an executable object file; give errors for undefined references. This is the default behavior for static mode. `-a` may not be used with the `-r` option.

`-b` In dynamic mode only, when creating an executable, do not do special processing for relocations that reference symbols in shared objects. Without the `-b` option, the link editor will create special position-independent relocations for references to functions defined in shared objects and will arrange for data objects defined in shared objects to be copied into the memory image of the executable by the dynamic linker at run time. With the `-b` option, the output code may be more efficient, but it will be less sharable.

`-d[y|n]` When `-dy`, the default, is specified, `ld` uses dynamic linking; when `-dn` is specified, `ld` uses static linking.

`-e` *epsym*
Set the entry point address for the output file to be that of the symbol *epsym*.

`-h` *name*
In dynamic mode only, when building a shared object, record *name* in the object's dynamic section. *name* will be recorded in executables that are linked with this object rather than the object's DG/UX system file name. Accordingly, *name* will be used by the dynamic linker as the name of the shared object to search for at run time.

`-l`*x* Search a library `lib`*x*`.so` or `lib`*x*`.a`, the conventional names for shared object and archive libraries, respectively. In dynamic mode, unless the `-Bstatic` option is in effect, `ld` searches each directory specified in the library search path for a file `libx.so` or `libx.a`. The directory search stops at the first directory containing either. `ld` chooses the file ending in `.so` if `-lx` expands to two files whose names are of the form `libx.so` and `libx.a`. If no `libx.so` is found, then `ld` accepts `libx.a`. In static mode, or when the `-Bstatic` option is in effect, `ld` selects only the file ending in `.a`. A library is searched when its name is encountered, so the placement of `-l` is significant.

`-m` Produce a memory map or listing of the input/output sections on the standard output.

`-o` *outfile*
: Produce an output object file named *outfile*. The name of the default object file is `a.out`.

`-r` Combine relocatable object files to produce one relocatable object file. `ld` will not complain about unresolved references. This option cannot be used in dynamic mode or with `-a`.

`-s` Strip symbolic information from the output file. The debug and line sections and their associated relocation entries will be removed. Except for relocatable files or shared objects, the symbol table and string table sections will also be removed from the output object file.

`-t` Turn off the warning about multiply defined symbols that are not the same size.

`-u` *symname*
: Enter *symname* as an undefined symbol in the symbol table. This is useful for loading entirely from an archive library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine. The placement of this option on the command line is significant; it must be placed before the library that will define the symbol.

`-z defs`
: Force a fatal error if any undefined symbols remain at the end of the link. This is the default when building an executable. It is also useful when building a shared object to assure that the object is self-contained, that is, that all its symbolic references are resolved internally.

`-z nodefs`
: Allow undefined symbols. This is the default when building a shared object. It may be used when building an executable in dynamic mode and linking with a shared object that has unresolved references in routines not used by that executable. This option should be used with caution.

`-z text`
: In dynamic mode only, force a fatal error if any relocations against non-writable, allocatable sections remain.

`-z` [`lowzeroes` | `lowzeros`]
: Support dereferencing of null pointers. The link editor creates a segment at addresses 0 (inclusive) through 0x1000 (exclusive), consisting entirely of read-only zeroes.

`-z sysinuser`
: Set the EF_88K_SYSINUSER flag in the executable file. This allows the operating system to place the process's stack and/or its dynamic segments in the user-managed area, provided they do not overlay any of the process's

loadable segments or its actual or potential break area.

-B [dynamic|static]
: Options governing library inclusion. -Bdynamic is valid in dynamic mode only. These options may be specified any number of times on the command line as toggles: if the -Bstatic option is given, no shared objects will be accepted until -Bdynamic is seen. See also the -l option.

-Bsymbolic
: In dynamic mode only, when building a shared object, bind references to global symbols to their definitions within the object, if definitions are available. Normally, references to global symbols within shared objects are not bound until run time, even if definitions are available, so that definitions of the same symbol in an executable or other shared objects can override the object's own definition. ld will issue warnings for undefined symbols unless -z defs overrides.

-G
: In dynamic mode only, produce a shared object. Undefined symbols are allowed by default (see -z defs, above).

-I *name*
: When building an executable, use *name* as the path name of the interpreter to be written into the program header. The default in static mode is no interpreter; in dynamic mode, the default is the name of the dynamic linker, /usr/lib/libc.so.1. Either case may be overridden by -I. exec will load this interpreter when it loads the a.out and will pass control to the interpreter rather than to the a.out directly.

-L *path*
: Add *path* to the library search directories. ld searches for libraries first in any directories specified with -L options, then in the standard directories. This option is effective only if it precedes the -l option on the command line.

-M *mapfile*
: In *static* mode only, read *mapfile* as a text file of directives to ld. Because these directives change the shape of the output file created by ld, use of this option is strongly discouraged.

-Q[y|n]
: Under -Qy, an ident string is added to the .comment section of the output file to identify the version of the link editor used to create the file. This will result in multiple ld idents when there have been multiple linking steps, such as when using ld -r. This is identical with the default action of the cc command. -Qn suppresses suppresses this behavior.

-V
: Output a message giving information about the version of ld being used.

-YP, *dirlist*
: Change the default directories used for finding libraries. *dirlist* is a colon-separated path list.

The environment variable LD_LIBRARY_PATH may be used to specify library search directories. In the most general case, it will contain two directory lists separated by a semicolon:

dirlist1 ; dirlist2

If ld is called with any number of occurrences of -L, as in ld ... -L*path1* ...-L*pathn* ...

then the search path ordering is

dirlist1 path1 ... pathn dirlist2 LIBPATH

LD_LIBRARY_PATH is also used to specify library search directories to the dynamic linker at run time. That is, if LD_LIBRARY_PATH exists in the environment, the dynamic linker will search the directories named in it, before its default directory, for shared objects to be linked with the program at execution.

The environment variable LD_RUN_PATH, containing a directory list, may also be used to specify library search directories to the dynamic linker. If present and not null, it is passed to the dynamic linker by ld via data stored in the output object file.

FILES

lib*x*.so	libraries
lib*x*.a	libraries
a.out	output file
LIBPATH	usually /usr/lib

SEE ALSO

as(1), cc(1), ld-coff(1), sde-target(1), exec(2), exit(2), end(3C), a.out(4), ar(4), sde(5).
The "C Compilation System" chapter and the "Mapfile Option" appendix in the *Programmer's Guide: ANSI C and Programming Support Tools*.

NOTES

Through its options, the link editor gives users great flexibility; however, those who use the -M *mapfile* option must assume some added responsibilities. Use of this feature is *strongly* discouraged.

NAME

ld – link editor for common object files

SYNOPSIS

ld [*options*] *filename* ... [*indirect-file* ...]

DESCRIPTION

The ld command combines several common object files into one, performs relocation, resolves external symbols, and supports symbol table information for symbolic debugging.

When given several object files, ld combines them, producing an executable object module. If you include the -r option on the command line, ld produces a linkable file (suitable for linking by another ld command) instead of an executable one. The output of ld is left in a.out by default. This file is executable if no errors occurred during the load. If any input file is not an object file, ld assumes it is either an archive library or an *indirect file,* a text file containing link editor directives. In an indirect file, one option letter, filename, or symbol assignment is put on each line. (See *Programmer's Guide: ANSI C and Programming Support Tools* for a discussion of input directives.)

If any argument is a library, the library is opened once, searched as many times as required, and then closed. Only those routines defining an unresolved external reference are loaded. Thus, library members can be in any order.

Options are:

-ansi
: On absolute links, do not produce the symbols etext, edata, or end, as these symbols are in the ansi namespace. The symbols __etext, __edata, and __end are still defined by the linker.

-e *symbol*
: Set the default entry point address for the output file to be that of the symbol *symbol*.

-f *fill*
: Set the default fill pattern for "holes" within an output section as well as initialized *bss* sections. The argument *fill* is a two-byte constant.

-F *magic*
: Give the program the magic number *magic,* in the conventional format for octal, decimal, or hexadecimal numbers. Octal numbers have a 0 prefix, hexadecimal numbers have an 0x prefix. Two magic numbers are valid for DG/UX: 0541 for DG/UX programs, 0555 for BCS compliant programs. The default magic number is 0541.

-l*name*
: Search for library lib*name*.a (*name* may be up to 9 characters in length). Ld searches in *LIBDIR* (usually /lib) and *LLIBDIR* (usually /usr/lib) by default. See -L. Note the format of the library name that ld searches for. This option must be specified on the command line after the object file names that contain references to a module in lib*name*.a.

-L *dir*
: Search for libraries in *dir* before searching *LIBDIR* and *LLIBDIR*. For this option to have any effect, you must also include the -l option. The -L option must precede the -l option on the command line.

-m
: Produce a link map.

-M
: Warn about multiply defined external definitions.

-N
: Put the text section at the beginning of the text segment rather than after all header information, and put the data section immediately

following text in the core image.

-n
: Do not make contributions to output sections that are not made by input files on the command line. Use of this option may cause the link to fail if dg/ux libraries or start up code are used. It will also prevent the linker from producing correct low level debugging information with input *.tdesc sections*. (See *Programmer's Guide: ANSI C and Programming Support Tools* for a discussion of ld handling of special sections.)

-o *file*
: Executable module is called *file* instead of `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. The link editor will not complain about unresolved references, and the output file will not be executable.

-a
: Create an absolute file. This is the default if the `-r` option is not used. Used with the `-r` option, `-a` allocates memory for common symbols.

-s
: Strip line number entries and symbol table information from the output object file.

-t
: Turn off the warning about multiply-defined symbols that are not the same size.

-u *name*
: Add *name* 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. The placement of this option on the `ld` command line is significant; it must be placed before the library which will define the symbol.

-V
: Print the revision number of `ld`.

-x
: Strip local symbols from the outputleave external and static symbols only. This option saves space in the output file.

-z
: Do not bind anything to address zero. This option will allow runtime detection of null pointers.

-Y [LU],*dir*
: Change the default directory used for finding libraries. If `L` is specified, the first default directory that `ld` searches, *LIBDIR*, is replaced by dir. If `U` is specified, the second default directory that `ld` searches, *LLIBDIR*, is replaced by *dir*. If `ld` was built with only one default directory but you specify `U` anyway, `ld` prints a warning and ignores the option.

FILES

LIBDIR/`libx.a`	libraries
LLIBDIR/`libx.a`	libraries
`a.out`	output file
LIBDIR	the first default search directory for libraries, usually `/lib`.
LLIBDIR	the second default search directory for libraries, usually */usr/lib*.

SEE ALSO

as(1), att_dump(1), cc(1), nm(1), size(1), exit(2), end(3C), a.out(4), ar(4).

Programmer's Guide: ANSI C and Programming Support Tools

CAVEATS

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 and options should ensure 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 program's address space.
- When the link editor is called through `cc(1)`, a startup routine is linked with the user's program. This routine calls `exit(2)` after execution of the main program. If the user calls the link editor directly, the user must insure that the program always calls `exit` rather than falling through the end of the entry routine.

The symbols *etext, edata,* and *end* [see `end(3c)`] are reserved and are defined by the link editor. It is incorrect for a user program to redefine them.

If the link editor does not recognize an input file as an object file or an archive file, it will assume that it contains link editor directives and will attempt to parse it. This will occasionally produce an error message compaining about "syntax errors."

Arithmetic expressions may have only one forward-referenced symbol per expression.

NAME

ldd – list dynamic dependencies

SYNOPSIS

`ldd [-d | -r]` *file*

DESCRIPTION

The `ldd` command lists the path names of all shared objects that would be loaded as a result of executing *file*. If *file* is a valid executable but does not require any shared objects, `ldd` will succeed, producing no output.

`ldd` may also be used to check the compatibility of *file* with the shared objects it uses. It does this by optionally printing warnings for any unresolved symbol references that would occur if *file* were executed. Two options govern this mode of `ldd`:

`-d` Causes `ldd` to check all references to data objects.

`-r` Causes `ldd` to check references to both data objects and functions.

Only one of the above options may be given during any single invocation of `ldd`.

DIAGNOSTICS

`ldd` prints its record of shared object path names to `stdout`. The optional list of symbol resolution problems are printed to `stderr`. If *file* is not an executable file or cannot be opened for reading, a non-zero exit status is returned.

SEE ALSO

`cc(1)`, `ld(1)`.
The "C Compilation System" chapter in the *Programmer's Guide: ANSI C and Programming Support Tools*.

NOTES

`ldd` doesn't list shared objects explicitly attached via `dlopen(3X)`.

`ldd` uses the same algorithm as the dynamic linker to locate shared objects.

NAME

`lex` – generate programs for simple lexical tasks

SYNOPSIS

`lex` [`-tvn`] [*file*] ...

DESCRIPTION

`Lex` generates programs to do simple lexical analysis of text using regular expressions. `Lex` reads its input *files*, or the standard input if no *files* are named, to get a list of regular expressions the generated program will look for, and C text to execute when each expression is matched.

An output file `lex.`*yy*`.c` is produced that contains C code for the generated program, which is named `yylex`. It must be linked using the `-ll` switch, to get the `lex` library routines.

The input to `lex` is of the form:

```
declarations
%%
rules
%%
programs
```

Any of the sections may be empty. If the "programs" section is empty, the "%%" that precedes it may be omitted. Thus the shortest legal `lex` input is

```
%%
```

Rules

Each rule is of the form:

<expression> *<action>*

An *<expression>* defines a regular expression that `yylex` will try to match. The *<action>* is the C code that `yylex` will execute when that *<expression>* is matched.

`yylex` writes any input characters that match no expression to the standard output.

The notation for `lex` regular expressions is described below. In the description, *X* and *Y* stand for `lex` regular expressions, and *x* and *y* stand for characters.

x An ordinary single character matches itself. Exceptions are these meta-characters: "\[]^-?.*+|()$/{}%<>.

\x Matches *x*, except for these special escape sequences beginning with a backslash:

- \n matches newline
- \t matches tab
- \b matches backspace
- \\ matches backslash

"*xy*" A string of characters in double quotes matches the string of characters. Any special meaning those characters (except for backslash) might otherwise have is ignored. The string "\x" matches whatever \x would match. For example,

- "." matches a period
- "\n" matches newline

"[hello]\t"
: matches the 8-character string "[hello]" followed by a tab

. A period matches any character except newline.

[*xy*] A string of elements inside square brackets matches any character any of the elements match. Elements can be any of the following:

single characters, which match themselves (except for "]" anywhere and "-" immediately after the initial "[").

\x regular expressions, which match what they usually do.

triplets of characters x-y; these match any character from x to y, inclusive. For example, [adm-p\n] matches any one of these characters: a, d, m, n, o, p, newline.

A caret, ^, as the first character inside the square brackets has special meaning: if S is a string of characters, then [^S] matches any character except for newline and any character that [S] would match.

XY matches anything that X would match concatenated with anything that Y would match. For example, `[ab][cd]` matches "ac", "bc", "ad", and "bd".

*X** matches 0 or more successive strings each matched by *X*. For example, `c*` matches the empty string, "c", "cc", and so forth.

X+ matches 1 or more successive strings each matched by *X*. For example, `c+` matches "c", "cc", and so forth.

X{j,k}
: where *j* and *k* are integers in the range [0,255], matches j to k (inclusive) successive strings each matched by X. For example, `c{3,5}` matches "ccc", "cccc", and "ccccc".

X{j} is equivalent to *X{j,j}*; it matches exactly j successive strings each matched by *X*.

X{j,} matches *j* or more successive strings matched by *X*.

(*X*) matches whatever *X* matches.

X? matches the empty string and whatever *X* matches; it is equivalent to *X*`{0,1}`. For example, `(ab)?` matches "ab" and "".

X|Y matches anything that either *X* or *Y* would match. For example, `"bob"|(ab?c)` matches "bob", "ac", and "abc".

^X A caret, ^, at the beginning of a regular expression restricts it to only match strings at the beginning of a line. A caret not at the beginning of a regular expression does not have this effect. For example, `^Bob` matches "Bob" when it occurs at the beginning of a line, but nowhere else.

X$ A dollar sign, `$`, at the end of a regular expression restricts it to only match strings at the end of a line. A dollar sign not at the end of a regular expression does not have this effect. For example, `bye$` matches "bye" when it occurs at the end of a line, but nowhere else.

X/Y restrict *X* to match only strings that are followed by something *Y* matches. For example, `(bob)/(white)` matches "bob" in the context "bobwhite" but not in the context "bobolink".

Blanks or tabs can only appear within a regular expression if each is:

- escaped with a backslash;
- inside double quotes; or
- within square brackets.

The <*action*> may be a single line of C code terminated with a semicolon, or a sequence of C statements within curly braces { and }. `Lex` provides the following for use in actions:

yytext Character pointer to the text matched by the regular expression.

yyleng Length of text in yytext.

| "|;" as the action for one rule is equivalent to the action for the next rule. "|" may not be used inside curly braces "{}".

ECHO Equivalent to

printf("%s", yytext)

REJECT
Causes `yylex` to reject this match and continue looking to see if other regular expressions will match it instead.

unput(c)
Routine that pushes a character back onto the input.

yyless(n)
Causes all but first n characters of yytext to be pushed back onto the input.

yymore()
Causes the next input string to be matched to be catenated onto the end of yytext, rather than overwriting it.

You can redefine several routines and macros to change how `yylex` behaves:

input() By default, a macro that is called to read a character from stdin. It returns 0 at end-of-file.

unput(c)
By default, a macro that is called to push the character c back onto the input. The `lex` library allows 100 characters worth of pushback.

If you redefine input() or unput(c), you must ensure that the two of them are consistent with each other.

output(c)
By default, a macro that is called to write a character c to stdout.

yyin File pointer for input; macro defined as stdin.

yyout File pointer for output; macro defined as stdout.

yywrap()
This routine is called when input() returns 0. If yywrap() returns 1, `yylex` finishes wrapping up and returns. If yywrap() returns 0, however, `yylex` continues to read input and match expressions. The default yywrap() always returns 1.

Declarations

The declarations section may contain:

- C code to be placed at the head of lex.yy.c. Any lines between lines containing only "%{" and "%}" are copied into lex.yy.c.
- Lex substitution string definitions. Each such definition is a line of the form:

```
name  definition
```

The name must start in the first column and begin with a letter, and it must be separated from the translation by one or more blanks or tabs. The translation can be anything.

Such names may be used in expressions in the rules section by surrounding them with curly braces, {}. For example,

```
DIGIT       [0-9]
%%
{DIGIT}+    printf("integer");
```

The "{DIGIT}" is replaced by its definition "[0-9]".

- Start condition definitions. Each definition line is of the form:

```
%Start cond1 cond2 ...
```

where the "%Start" begins in the first column. Each word following it is declared to be the name of a start condition.

Expressions in the rules section may then be preceded by the names of start conditions in angle brackets, <>; this restricts them to be matched only when yylex is in the listed start conditions. Several start conditions may be listed, separated by commas; for example, "<cond1,cond2>".

The start condition yylex is in may be changed by an action that executes a "BEGIN name;" statement, where "name" is the name of a start condition. yylex is initially in start condition 0, or INITIAL; "BEGIN 0;" or "BEGIN INITIAL;" will reset it.

NOTE: Any expression not preceded by a start condition may be matched at any time. For example,

```
%Start     one two
%%
^one              { ECHO; BEGIN one; }
^two              { ECHO; BEGIN two; }
^zip              { ECHO; BEGIN zip; }
onetarget   { printf("one"); }
twotarget   { printf("two"); }
```

Different rules for "target" will be executed depending on what start condition is active.

- Table size limits for the finite state machine implemented by yylex.

```
%p n    Maximum number of positions is n (default 2000)
%n n    Maximum number of states is n (500)
```

```
%t n    Maximum number of parse tree nodes is n (1000)
%a n    Maximum number of transitions is n (3000)
```

Programs

The programs section may contain anything you like. It is copied to the end of `lex.yy.c`.

Any line in any of the three sections that begins with a space is copied directly into `lex.yy.c`.

To use `yylex`, you must provide a program to call it and link them with the "-ll" option. To use `yylex` with a `yacc(1)` parser, end the action for each `lex` rule with

```
return(token);
```

where "token" is the appropriate token. Access to *yacc*'s token names may be ensured by including the `yylex` code in the *yacc* generator with

```
#include "lex.yy.c"
```

or generating the "y.tab.h" file with yacc's "-d" option and including it with

```
#include "y.tab.h"
```

in the definitions section of the `lex` input.

Options

- `-t` Output which normally goes to lex.yy.c is sent to stdout.
- `-v` A one-line summary of the finite state machine implemented by `yylex` is printed.
- `-n` Cancels -v option.

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");
"/*" {       loop:
                while (input() != '*');
                switch (input())
                        {
                        case '/': break;
                        case '*': unput('*');
                        default: go to loop;
                        }
                }
```

SEE ALSO

yacc(1), malloc(3X).

NAME

lint - a C program checker

SYNOPSIS

lint [*option* ...] *file* ...

DESCRIPTION

Lint 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 does the compiler. Lint detects unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. It also checks for 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 used but not 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 generate warnings and are ignored.

Lint processes all the .c, .ln, and llib-l*x*.ln (specified by -l*x*) files 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. (In a COFF environment, the -p option causes the portable C lint library (llib-port.ln) to be 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.

OPTIONS

Lint is sensitive to the target environment (see sde-target(1) and sde(5)): lint options that are accepted only in an ELF target environment are noted below.

Lint recognizes many cc(1) and cpp(1) command line options, including -D, -U, -g, and -O, although -g and -O are ignored. In an ELF target environment, the cc options -Xa, -Xc, and -Xt can be used to indicate to lint the degree of ANSI conformance to be found in the source. When coding to the ANSI C standard in a COFF target environment, you may wish to run lint in an ELF environment with the appropriate option.

You can use any number of lint options, in any order, intermixed with file-name arguments. The following options 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 try to find 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.

-m Suppress complaints about external symbols that could be declared static (ELF environment only).

The following arguments alter `lint`'s behavior. Some of these option settings are also available through `lint` comments (see below).

-l*x* Include the 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. You can use this option to reference local `lint` libraries and to develop multi-file projects.

-L*directory*
Look in *directory* first for libraries, then go to `/usr/lib` libraries not found in *directory*. You can specify several directories by giving the *-L* option and a directory name for each directory you want searched.

-n Do not check compatibility against either the standard or the portable `lint` library.

-p Check portability to other dialects 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 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 the `-o` option. The `lint` library produced is the input that is given to `lint`'s second pass. The `-o` option simply saves this file in the named lint library. To produce a `llib-l`*lib*`.ln` without extraneous messages, use the `-x` option. 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 arguments are accepted only in an ELF environment:

-I*dir* Search for included header files in the directory *dir* before searching the current directory and/or the standard place.

-s Produce one-line diagnostics only. `lint` occasionally buffers messages to produce a compound report.

-k Alter the behavior of /*LINTED [*message*]*/ directives. Normally, `lint` will suppress warning messages for the code following these directives. Instead of suppressing the messages, `lint` prints an additional message containing the comment inside the directive.

-y Specify that the file being linted will be treated as if the /*LINTLIBRARY*/ directive had been used. A lint library is normally created by using the /*LINTLIBRARY*/ directive.

-F Print pathnames of files. `lint` normally prints the filename without the path.

-V Write to standard error the product name and release.

-W*file* Write a `.ln` file to *file*, for use by `cflow`(1).

`-R`*file* Write a `.ln` file to *file*, for use by `cxref`(1).

Unrecognized options are warned about and ignored. The predefined macro `lint` is defined to allow certain questionable code to be altered or removed for `lint`. Thus, 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*/` 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. Equivalent to using the `-v` and `-x` options.

These comments are recognized only in an ELF environment:

`/*CONSTCOND*/ or /*CONSTANTCOND*/ or /*CONSTANTCONDITION*/`
suppresses complaints about constant operands for the next expression.

`/*EMPTY*/` suppresses complaints about a null statement consequent on an if statement. This directive should be placed after the test expression, and before the semicolon. This directive is supplied to support empty if statements when a valid else statement follows. It suppresses messages on an empty `else` consequent.

`/*FALLTHRU*/ or /*FALLTHROUGH*/`
suppresses complaints about fall through to a `case` or `default` labelled statement. This directive should be placed immediately preceding the label.

`/*LINTED [`*message*`]*/`
suppresses any intra-file warning except those dealing with unused variables or functions. This directive should be placed on the line immediately preceding where the lint warning occurred. The `-k` option alters the way in which `lint` handles this directive. Instead of suppressing messages, `lint` will print an additional message, if any, contained in the comment. This directive is useful in conjunction with the `-s` option for post-lint filtering.

`/*PRINTFLIKE`*n*`*/` makes `lint` check the first *(n-1)* arguments as usual. The *nth* argument is interpreted as a `printf` format string that is used to check the remaining arguments.

`/*PROTOLIB`*n*`*/` causes `lint` to treat function declaration prototypes as function definitions if *n* is non-zero. This directive can be used only in conjunction with the
`/* LINTLIBRARY */` directive. If *n* is zero, function prototypes will be treated normally.

`/*SCANFLIKE`*n*`*/` makes `lint` check the first *(n-1)* arguments as usual. The *nth* argument is interpreted as a `scanf` format string that is used to check the remaining arguments.

Lint produces its output in three phases. In the first, it prints messages for each source file. In the second phase, it prints messages for any files included with #include. In the final phase, it prints messages about interrelations between files. Question marks after filenames in this phase indicate lint could not determine exactly what file the message refers to. The third phase is not done if the -c is given.

The behavior of the -c and the -o options allows for incremental use of lint on a set of C source files. You can invoke lint once for each source file with the -c option. Each of these invocations produces a .ln file that corresponds to the .c file, and prints all messages about that source file only. After .ln files have been produced, for all the source files, lint 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 lets you use make to lint only the source files that have been modified since the last time the set of source files were linted.

FILES

/usr/lib	The directory where the lint libraries specified by the -l*x* option must exist; usually /usr/lib
LIBDIR/lint[12]	First and second passes
LIBDIR/llib-lc.ln	Declarations for C Library functions (binary format; if you have bought a source license, the source is in *LIBDIR/llib-lc*)
LIBDIR/llib-port.ln	Declarations for portable functions (binary format; if you have a source license, the source is in *LIBDIR/llib-port*)
LIBDIR/llib-lm.ln	Declarations for Math Library functions (binary format; if you have a source license, the source is in *LIBDIR/llib-lm*)
/usr/tmp/*lint*	Temporaries

SEE ALSO

cc(1), cpp(1), make(1), sde-target(1), sde(5),
Programmer's Guide: ANSI C and Programming Support Tools.

NOTES

Exit(2), longjmp(3C), and other functions that do not return are not understood; this causes various problems.

NAME

lorder – find ordering relation for an object library

SYNOPSIS

lorder *file* ...

DESCRIPTION

Lorder reads each object or archive *file* and produces a list of pairs of object file or archive member names. The first file of each pair refers to external identifiers defined in the second file.

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) can perform 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. Using lorder(1) may, however, allow for a slightly more efficient access of the archive during the link edit process.

The following example builds a new library from existing object files:

```
ar cr library `lorder *.o | tsort`
```

FILES

TMPDIR/*symref
: Temporary file of symbol references

TMPDIR/*symdef
: Temporary file of symbol definitions

TMPDIR
: TMPDIR is usually /usr/tmp but can be redefined by setting the environment variable TMPDIR [see *tmpnam()*] 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.

NAME

m4 – macro processor

SYNOPSIS

m4 [*options*] [*files*]

DESCRIPTION

M4 is a macro processor intended as a front end for C and other languages. Each of the argument files is processed in order; if there are no files, or if a file name is –, the standard input is read. The processed text is written on the standard output.

The options and their effects are as follows:

–e Operate interactively. Interrupts are ignored and the output is unbuffered.

–s Enable line sync output for the C preprocessor (#line ...)

–B*int* Change the size of the push-back and argument collection buffers from the default of 4096.

–H*int* Change the size of the symbol table hash array from the default of 199. The size should be prime.

–S*int* Change the size of the call stack from the default size of 100 slots. Macros take three slots, and non-macro arguments take one.

–T*int* Change the size of the token buffer from the default of 512 bytes.

To be effective, these flags must appear before any file names and before any –D or –U flags:

–D*name*[=*val*]
Defines *name* to *val* or to null in *val*'s absence.

–U*name*
undefines *name*.

Macro calls have the form:

name(arg1,arg2, ..., argn)

The (must immediately follow the name of the macro. If the name of a defined macro is not followed by a (, it is deemed to be a call of that macro with no arguments. Potential macro names consist of alphabetic letters, digits, and underscore (_), where the first character is not a digit.

Leading unquoted blanks, tabs, and new-lines are ignored while collecting arguments. Left and right single quotes are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. If fewer arguments are supplied than are in the macro definition, the trailing arguments are taken to be null. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses which happen to turn up within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

M4 makes available the following built-in macros. They may be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.

define the second argument is installed as the value of the macro whose name is the first argument. Each occurrence of $*n* in the replacement text, where *n* is a digit, is replaced by the *n*-th argument. Argument 0 is the name of the macro; missing arguments are replaced by the null string; `$#` is replaced by the number of arguments; `$*` is replaced by a list of all the arguments separated by commas; `$@` is like `$*`, but each argument is quoted (with the current quotes).

undefine — removes the definition of the macro named in its argument.

defn — returns the quoted definition of its argument(s). It is useful for renaming macros, especially built-ins.

pushdef — like *define*, but saves any previous definition.

popdef — removes current definition of its argument(s), exposing the previous one, if any.

ifdef — if the first argument is defined, the value is the second argument, otherwise the third. If there is no third argument, the value is null. The word *unix* is predefined on UNIX system versions of `m4`.

shift — returns all but its first argument. The other arguments are quoted and pushed back with commas in between. The quoting nullifies the effect of the extra scan that will subsequently be performed.

changequote — change quote symbols to the first and second arguments. The symbols may be up to five characters long. *Changequote* without arguments restores the original values (i.e., `').

changecom — change left and right comment markers from the default # and new-line. With no arguments, the comment mechanism is effectively disabled. With one argument, the left marker becomes the argument and the right marker becomes new-line. With two arguments, both markers are affected. Comment markers may be up to five characters long.

divert — `m4` maintains 10 output streams, numbered 0-9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. The *divert* macro changes the current output stream to its (digit-string) argument. Output diverted to a stream other than 0 through 9 is discarded.

undivert — causes immediate output of text from diversions named as arguments, or all diversions if no argument. Text may be undiverted into another diversion. Undiverting discards the diverted text.

divnum — returns the value of the current output stream.

dnl — reads and discards characters up to and including the next new-line.

ifelse — has three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6 and 7. Otherwise, the value is either the fourth string, or, if it is not present, null.

incr — returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

decr — returns the value of its argument decremented by 1.

eval — evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include `+`, `-`, `*`, `/`, `%`, `^` (exponentiation), bitwise `&`, `|`, `^`, and `~`; relationals; parentheses. Octal and hex numbers may be specified as in C. The second argument specifies the radix for the result; the default is 10. The third argument may be used to specify the minimum number of digits in the result.

len returns the number of characters in its argument.

index returns the position in its first argument where the second argument begins (zero origin), or −1 if the second argument does not occur.

substr returns a substring of its first argument. The second argument is a zero origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.

translit transliterates the characters in its first argument from the set given by the second argument to the set given by the third. No abbreviations are permitted.

include returns the contents of the file named in the argument.

sinclude is identical to *include*, except that it says nothing if the file is inaccessible.

syscmd executes the DG/UX system command given in the first argument. No value is returned.

sysval is the return code from the last call to *syscmd*.

maketemp fills in a string of XXXXXX at the end of its argument with a unique letter and the current process ID.

m4exit causes immediate exit from `m4`. Argument 1, if given, is the exit code; the default is 0.

m4wrap argument 1 will be pushed back at final EOF so that it gets evaluated Example: `m4wrap('cleanup()')`

errprint prints its argument on the diagnostic output file.

dumpdef prints current names and definitions, for the named items, or for all if no arguments are given.

traceon with no arguments, turns on tracing for all macros (including built-ins). Otherwise, turns on tracing for named macros.

traceoff turns off trace globally and for any macros specified. Macros specifically traced by *traceon* can be untraced only by specific calls to *traceoff*.

EXAMPLE

```
m4 file1 file2 > outputfile
```

will run the `m4` macro processor on the files `file1` and `file2`, redirecting the output into `outputfile`.

SEE ALSO

cc(1), cpp(1).

The M4 Macro Processor by B. W. Kernighan and D. M. Ritchie.

NAME

make – maintain, update, and regenerate groups of programs

SYNOPSIS

make [-f *makefile*] [-eiknpqrst] [*names*]

DESCRIPTION

make allows the programmer to maintain, update, and regenerate groups of computer programs. make executes commands in *makefile* to update one or more target *names* (*names* are typically programs). If the -f option is not present, then makefile, Makefile, and the Source Code Control System (SCCS) files s.makefile, and s.Makefile are tried in order. If *makefile* is -, the standard input is taken. More than one -f *makefile* argument pair may appear.

make updates a target only if its dependents are newer than the target. All prerequisite files of a target are added recursively to the list of targets. Missing files are deemed to be outdated.

The following list of four directives can be included in *makefile* to extend the options provided by make. They are used in *makefile* as if they were targets:

.DEFAULT: If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name .DEFAULT are used if it exists.

.IGNORE: Same effect as the -i option.

.PRECIOUS: Dependents of the .PRECIOUS entry will not be removed when quit or interrupt are hit.

.SILENT: Same effect as the -s option.

The options for make are listed below:

-e Environment variables override assignments within makefiles.

-f *makefile* Description filename (*makefile* is assumed to be the name of a description file).

-i Ignore error codes returned by invoked commands.

-k Abandon work on the current entry if it fails, but continue on other branches that do not depend on that entry.

-n No execute mode. Print commands, but do not execute them. Even command lines beginning with an @ are printed.

-p Print out the complete set of macro definitions and target descriptions.

-q Question. make returns a zero or non-zero status code depending on whether or not the target file has been updated.

-r Do not use the built-in rules.

-s Silent mode. Do not print command lines before executing.

-t Touch the target files (causing them to be updated) rather than issue the usual commands.

Creating the makefile

The makefile invoked with the -f option is a carefully structured file of explicit instructions for updating and regenerating programs, and contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, non-null list of targets, then a :, then a (possibly null) list of prerequisite files or dependencies. Text following a ; and all following lines that begin with a tab are shell commands to be executed to update the target. The first non-empty line that does not begin with a tab or # begins a new dependency or macro definition. Shell commands may be continued across lines with a backslash-new-line (\ new-line) sequence. Everything printed by make (except the initial tab) is passed directly to the shell as is. Thus,

```
echo a\
b
```

will produce

```
ab
```

exactly the same as the shell would.

Sharp (#) and new-line surround comments including contained \ new-line sequences.

The following makefile says that `pgm` depends on two files `a.o` and `b.o`, and that they in turn depend on their corresponding source files (`a.c` and `b.c`) and a common file `incl.h`:

```
pgm: a.o b.o
        cc a.o b.o -o pgm
a.o: incl.h a.c
        cc -c a.c
b.o: incl.h b.c
        cc -c b.c
```

Command lines are executed one at a time, each by its own shell. The `SHELL` environment variable can be used to specify which shell `make` should use to execute commands. The default is `/usr/bin/sh`. The first one or two characters in a command can be the following: @, -, @-, or -@. If @ is present, printing of the command is suppressed. If - is present, `make` ignores an error. A line is printed when it is executed unless the `-s` option is present, or the entry `.SILENT:` is included in *makefile*, or unless the initial character sequence contains a @. The `-n` option specifies printing without execution; however, if the command line has the string `$(MAKE)` in it, the line is always executed (see the discussion of the `MAKEFLAGS` macro in the "Environment" section below). The `-t` (touch) option updates the modified date of a file without executing any commands.

Commands returning non-zero status normally terminate `make`. If the `-i` option is present, if the entry `.IGNORE:` is included in *makefile*, or if the initial character sequence of the command contains -, the error is ignored. If the `-k` option is present, work is abandoned on the current entry, but continues on other branches that do not depend on that entry.

Interrupt and quit cause the target to be deleted unless the target is a dependent of the directive `.PRECIOUS`.

Environment

The environment is read by `make`. All variables are assumed to be macro definitions and are processed as such. The environment variables are processed before any makefile and after the internal rules; thus, macro assignments in a makefile override environment variables. The `-e` option causes the environment to override the macro assignments in a makefile. Suffixes and their associated rules in the makefile will override any identical suffixes in the built-in rules.

The MAKEFLAGS environment variable is processed by make as containing any legal input option (except -f and -p) defined for the command line. Further, upon invocation, make "invents" the variable if it is not in the environment, puts the current options into it, and passes it on to invocations of commands. Thus, MAKEFLAGS always contains the current input options. This feature proves very useful for "super-makes". In fact, as noted above, when the -n option is used, the command $(MAKE) is executed anyway; hence, one can perform a make -n recursively on a whole software system to see what would have been executed. This result is possible because the -n is put in MAKEFLAGS and passed to further invocations of $(MAKE). This usage is one way of debugging all of the makefiles for a software project without actually doing anything.

Include Files

If the string include appears as the first seven letters of a line in a *makefile*, and is followed by a blank or a tab, the rest of the line is assumed to be a filename and will be read by the current invocation, after substituting for any macros.

Macros

Entries of the form *string1* = *string2* are macro definitions. *string2* is defined as all characters up to a comment character or an unescaped new-line. Subsequent appearances of $(*string1*[:*subst1*=[*subst2*]]) are replaced by *string2*. The parentheses are optional if a single-character macro name is used and there is no substitute sequence. The optional :*subst1*=*subst2* is a substitute sequence. If it is specified, all non-overlapping occurrences of *subst1* in the named macro are replaced by *subst2*. Strings (for the purposes of this type of substitution) are delimited by blanks, tabs, new-line characters, and beginnings of lines. An example of the use of the substitute sequence is shown in the "Libraries" section below.

Internal Macros

There are five internally maintained macros that are useful for writing rules for building targets.

$* The macro $* stands for the filename part of the current dependent with the suffix deleted. It is evaluated only for inference rules.

$@ The $@ macro stands for the full target name of the current target. It is evaluated only for explicitly named dependencies.

$< The $< macro is only evaluated for inference rules or the .DEFAULT rule. It is the module that is outdated with respect to the target (the "manufactured" dependent file name). Thus, in the .c.o rule, the $< macro would evaluate to the .c file. An example for making optimized .o files from .c files is:

```
.c.o:
        cc -c -O $*.c
```

or:

```
.c.o:
        cc -c -O $<
```

$? The $? macro is evaluated when explicit rules from the makefile are evaluated. It is the list of prerequisites that are outdated with respect to the target, and essentially those modules that must be rebuilt.

$% The $% macro is only evaluated when the target is an archive library member of the form lib(file.o). In this case, $@ evaluates to lib and $% evaluates to the library member, file.o.

Four of the five macros can have alternative forms. When an upper case D or F is appended to any of the four macros, the meaning is changed to "directory part" for

D and "file part" for F. Thus, $(@D) refers to the directory part of the string $@. If there is no directory part, ./ is generated. The only macro excluded from this alternative form is $?.

Suffixes

Certain names (for instance, those ending with .o) have inferable prerequisites such as .c, .s, etc. If no update commands for such a file appear in *makefile*, and if an inferable prerequisite exists, that prerequisite is compiled to make the target. In this case, make has inference rules that allow building files from other files by examining the suffixes and determining an appropriate inference rule to use. The current default inference rules are:

.c	.c~	.f	.f~	.s	.s~	.sh	.sh~	.C	.C~
.c.a	.c.o	.c~.a	.c~.c	.c~.o	.f.a	.f.o	.f~.a	.f~.f	.f~.o
.h~.h	.l.c	.l.o	.l~.c	.l~.l	.l~.o	.s.a	.s.o	.s~.a	.s~.o
.s~.s	.sh~.sh	.y.c	.y.o	.y~.c	.y~.o	.y~.y	.C.a	.C.o	.C~.a
.C~.C	.C~.o	.L.C	.L.o	.L~.C	.L~.L	.L~.o	.Y.C	.Y.o	.Y~.C
.Y~.o	.Y~.Y								

The internal rules for make are contained in the source file rules.c for the make program. These rules can be locally modified. To print out the rules compiled into the make on any machine in a form suitable for recompilation, the following command is used:

```
make -pf - 2>/dev/null </dev/null
```

A tilde in the above rules refers to an SCCS file [see sccsfile(4)]. Thus, the rule .c~.o would transform an SCCS C source file into an object file (.o). Because the s. of the SCCS files is a prefix, it is incompatible with the make suffix point of view. Hence, the tilde is a way of changing any file reference into an SCCS file reference.

A rule with only one suffix (for example, .c:) is the definition of how to build *x* from *x*.c. In effect, the other suffix is null. This feature is useful for building targets from only one source file, for example, shell procedures and simple C programs.

Additional suffixes are given as the dependency list for .SUFFIXES. Order is significant: the first possible name for which both a file and a rule exist is inferred as a prerequisite. The default list is:

```
.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .sh .sh~ .h .h~ .f .f~
.C .C~ .Y .Y~ .L .L~
```

Here again, the above command for printing the internal rules will display the list of suffixes implemented on the current machine. Multiple suffix lists accumulate; .SUFFIXES: with no dependencies clears the list of suffixes.

Inference Rules

The first example can be done more briefly.

```
pgm: a.o b.o
     cc a.o b.o -o pgm
a.o b.o: incl.h
```

This abbreviation is possible because make has a set of internal rules for building files. The user may add rules to this list by simply putting them in the *makefile*.

Certain macros are used by the default inference rules to permit the inclusion of optional matter in any resulting commands. For example, CFLAGS, LFLAGS, and YFLAGS are used for compiler options to cc(1), lex(1), and yacc(1), respectively. Again, the previous method for examining the current rules is recommended.

The inference of prerequisites can be controlled. The rule to create a file with suffix `.o` from a file with suffix `.c` is specified as an entry with `.c.o:` as the target and no dependents. Shell commands associated with the target define the rule for making a `.o` file from a `.c` file. Any target that has no slashes in it and starts with a dot is identified as a rule and not a true target.

Libraries

If a target or dependency name contains parentheses, it is assumed to be an archive library, the string within parentheses referring to a member within the library. Thus, `lib(file.o)` and `$(LIB)(file.o)` both refer to an archive library that contains `file.o`. (This example assumes the `LIB` macro has been previously defined.) The expression `$(LIB)(file1.o file2.o)` is not legal. Rules pertaining to archive libraries have the form `.`*XX*`.a` where the *XX* is the suffix from which the archive member is to be made. An unfortunate by-product of the current implementation requires the *XX* to be different from the suffix of the archive member. Thus, one cannot have `lib(file.o)` depend upon `file.o` explicitly. The most common use of the archive interface follows. Here, we assume the source files are all C type source:

```
lib: lib(file1.o) lib(file2.o) lib(file3.o)
     @echo lib is now up-to-date
.c.a:
     $(CC) -c $(CFLAGS) $<
     $(AR) $(ARFLAGS) $@ $*.o
     rm -f $*.o
```

In fact, the `.c.a` rule listed above is built into `make` and is unnecessary in this example. A more interesting, but more limited example of an archive library maintenance construction follows:

```
lib: lib(file1.o) lib(file2.o) lib(file3.o)
     $(CC) -c $(CFLAGS) $(?:.o=.c)
     $(AR) $(ARFLAGS) lib $?
     rm $?
     @echo lib is now up-to-date
.c.a:;
```

Here the substitution mode of the macro expansions is used. The `$?` list is defined to be the set of object filenames (inside `lib`) whose C source files are outdated. The substitution mode translates the `.o` to `.c`. (Unfortunately, one cannot as yet transform to `.c~`; however, this transformation may become possible in the future.) Also note the disabling of the `.c.a:` rule, which would have created each object file, one by one. This particular construct speeds up archive library maintenance considerably. This type of construct becomes very cumbersome if the archive library contains a mix of assembly programs and C programs.

FILES

[Mm]akefile and **s.[Mm]akefile**
/usr/bin/sh

SEE ALSO

cc(1), **lex**(1), **yacc**(1), **printf**(3S), **sccsfile**(4).

cd(1), **sh**(1) in the *User's Reference Manual*.

See the "make" chapter in the *Programmer's Guide: ANSI C and Programming Support Tools*.

NOTES

Some commands return non-zero status inappropriately; use `-i` or the `-` command line prefix to overcome the difficulty.

Filenames with the characters = : @ will not work. Commands that are directly executed by the shell, notably cd(1), are ineffectual across new-lines in `make`. The syntax `lib(file1.o file2.o file3.o)` is illegal. You cannot build `lib(file.o)` from `file.o`.

NAME

mcs – manipulate the comment section of an object file.

SYNOPSIS

mcs [-a *string*] [-c] [-d] [-n *name*] [-p] [-V] *file* ...

DESCRIPTION

mcs is used to manipulate a section, by default the .comment section, in an ELF object file. It is used to add to, delete, print, and compress the contents of a section in an ELF object file; it can only print the contents of a section in a COFF object file. mcs must be given one or more of the options described below. It applies each of the options in order to each file.

These options are available:

-a *string*
: Append *string* to the comment section of the ELF object files. If *string* contains embedded blanks, it must be enclosed in quotation marks.

-c
: Compress the contents of the comment section of the ELF object files. All duplicate entries are removed. The ordering of the remaining entries is not disturbed.

-d
: Delete the contents of the comment section from the ELF object files. The section header for the comment section is also removed.

-n *name*
: Specify the name of the comment section to access if other than .comment. By default, mcs deals with the section named .comment. This option can be used to specify another section.

-p
: Print the contents of the comment section on the standard output. Each section printed is tagged by the name of the file from which it was extracted, using the format *filename*[*member_name*]: for archive files; and *filename*: for other files.

-V
: Print, on standard error, the version number of mcs.

If the input file is an archive [see ar(4)], the archive is treated as a set of individual files. For example, if the -a option is specified, the string is appended to the comment section of each ELF object file in the archive; if the archive member is not an ELF object file; then it is left unchanged.

If mcs is executed on an archive file, the archive symbol table will be removed unless -p is the only option specified. The archive symbol table must be restored by executing the ar command with the -ts option before the archive can be linked by the ld command. mcs will produce appropriate warning messages when this situation arises.

EXAMPLES

```
mcs -p file          # Print file's comment section

mcs -a string file   # Append string to file's comment section
```

FILES

TMPDIR/mcs*
: temporary files

TMPDIR
: usually /usr/tmp but can be redefined by setting the environment variable TMPDIR [see tempnam() in tmpnam(3S)].

SEE ALSO

ar(1), as(1), cc(1), ld(1), tmpnam(3S), a.out(4), ar(4).

See the "Object Files" chapter in *Programmer's Guide: ANSI C and Programming Support Tools*.

NOTES

mcs cannot add to, delete or compress the contents of a section that is contained within a segment.

NAME

mkstr – create an error message file by massaging C source

SYNOPSIS

mkstr [–] *messagefile prefix file* ...

DESCRIPTION

Mkstr is used to create files of error messages. Its use can make programs with large numbers of error diagnostics much smaller, and reduce system overhead in running the program as the error messages do not have to be constantly swapped in and out.

Mkstr will process each of the specified *files*, placing a massaged version of the input file in a file whose name consists of the specified *prefix* and the original name. A typical usage of mkstr would be

mkstr pistrings xx *.c

This command would cause all the error messages from the C source files in the current directory to be placed in the file *pistrings* and processed copies of the source for these files to be placed in files whose names are prefixed with *xx*.

To process the error messages in the source to the message file mkstr keys on the string 'error("' in the input stream. Each time it occurs, the C string starting at the '"' is placed in the message file followed by a null character and a new-line character; the null character terminates the message so it can be easily used when retrieved, the new-line character makes it possible to sensibly cat the error message file to see its contents. The massaged copy of the input file then contains an lseek pointer into the file which can be used to retrieve the message, i.e.:

```
char  efilname[] =  "/usr/lib/pistrings";
int   efil = -1;

error(a1, a2, a3, a4)
{
      char buf[256];

      if (efil < 0) {
            efil = open(efilname, 0);
            if (efil < 0) {
oops:
                  perror(efilname);
                  exit(1);
            }
      }
      if (lseek(efil, (long) a1, 0) || read(efil, buf, 256) <= 0)
            goto oops;
      printf(buf, a2, a3, a4);
}
```

The optional – causes the error messages to be placed at the end of the specified message file for recompiling part of a large mkstr-ed program.

EXAMPLE

If the current directory has files "a.c" and "b.c", then

```
mkstr exs x *.c
```

would create a new file "exs" which holds all the error messages extracted from the source files "a.c" and "b.c", as well as two new source files, "xa.c" and "xb.c", which no longer contain the extracted error messages.

SEE ALSO

`lseek`(2), `xstr`(1).

AUTHORS

William Joy and Charles Haley

NAME

nm – print name list of common object file

SYNOPSIS

nm [-oxhvnefurplVT] *file* ...

DESCRIPTION

The nm command displays the symbol table of each ELF or COFF object *file* specified. The file may be a relocatable or absolute ELF or COFF object file, or it may be an archive of relocatable or absolute ELF or COFF object files.

The information reported by nm differs in the ELF and COFF environments. In an ELF environment, nm prints the following:

`Index` The index of the symbol (the index appears in brackets).

`Value` The value of the symbol is one of the following: a section offset for defined symbols in a relocatable file; alignment constraints for symbols whose section index is `SHN_COMMON`; a virtual address in executable and dynamic library files.

`Size` The size in bytes of the associated object.

`Type` A symbol is of one of the following types: `NOTYPE` (no type was specified), `OBJECT` (a data object such as an array or variable), `FUNC` (a function or other executable code), `SECTION` (a section symbol), or `FILE` (name of the source file).

`Bind` The symbol's binding attributes. `LOCAL` symbols have a scope limited to the object file containing their definition; `GLOBAL` symbols are visible to all object files being combined; and `WEAK` symbols are essentially global symbols with a lower precedence than `GLOBAL`.

`Other` A field reserved for future use, currently containing 0.

`Shndx` Except for three special values, this is the section header table index in relation to which the symbol is defined. The following special values exist: `ABS` indicates the symbol's value will not change through relocation; `COMMON` indicates an unallocated block and the value provides alignment constraints; and `UNDEF` indicates an undefined symbol.

`Name` The name of the symbol.

In a COFF environment, nm prints the following:

`Name` The name of the symbol.

`Value` The symbol's value expressed as an offset or an address depending on its storage class.

`Class` The symbol's storage class.

`Type` The symbol's type and derived type. If the symbol is an instance of a structure or of a union then the structure or union tag will be given following the type (e.g., *struct-tag*). If the symbol is an array, then the array dimensions will be given following the type (e.g., char[n][m]). Note that the file must have been compiled with the −g option of the cc(1) command for this information to appear.

`Size` The symbol's size in bytes, if available. Note that the file must have been compiled with the −g option of the cc(1) command for this information to appear.

`Line` The source line number at which the symbol is defined, if available. Note that the file must have been compiled with the `-g` option of the `cc`(1) command for this information to appear.

`Section` For storage classes static and external, the object file section containing the symbol (e.g., text, data or bss).

These options control the output of `nm`:

`-o` Print the value and size of a symbol in octal instead of decimal.

`-x` Print the value and size of a symbol in hexadecimal instead of decimal.

`-h` Do not display the output header data.

`-v` Sort external symbols by value before they are printed.

`-n` Sort external symbols by name before they are printed.

`-u` Print undefined symbols only.

`-r` Prepend the name of the object file or archive to each output line.

`-p` Produce easily parsable, terse output. Each symbol name is preceded by its value (blanks if undefined) and one of the letters `U` (undefined), `A` (absolute), `T` (text symbol), `D` (data symbol), `S` (section symbol), `R` (register symbol), `F` (file symbol), `C` (common symbol), or `N` (symbol has no type). If the symbol is local (non-external), the type letter is in lower case.

`-V` Print, on standard error, the version of *nm* being executed.

This option is accepted only in an ELF environment:

`-l` Distinguish between `WEAK` and `GLOBAL` symbols by appending a * to the key letter for `WEAK` symbols.

These options are meaningful only in a COFF environment (they are ignored in an ELF environment):

`-e` Print only external and static symbols.

`-f` Produce full output. Print redundant symbols (*.text*, *.data*, *.lib*, and *.bss*), which are normally suppressed.

`-T` By default, `nm` prints the entire name of each symbol. Since object files can have symbol names with an arbitrary number of characters, a name that is longer than the width of the column set aside for names will overflow its column, forcing every column after the name to be misaligned. The `-T` option causes `nm` to truncate every name which would otherwise overflow its column and place an asterisk as the last character in the displayed name to mark it as truncated.

Options may be used in any order, either singly or in combination, and may appear anywhere in the command line. Therefore, both `nm name -e -v` and `nm -ve name` print the static and external symbols in *name*, with external symbols sorted by value.

FILES

*TMPDIR/** temporary files

TMPDIR is usually `/usr/tmp` but can be redefined by setting the environment variable `TMPDIR` [see `tmpnam()` in `tmpnam`(3S)].

DIAGNOSTICS

"nm: *name*: cannot open"
: if *name* cannot be read.

"nm: *name*: bad magic"
: if *name* is not a common object file.

"nm: *name*: no symbols"
: if the symbols have been stripped from *name*.

SEE ALSO

as(1), cc(1), ld(1), tmpnam(3S), a.out(4), ar(4).

CAVEAT

When all the symbols are printed, they must be printed in the order they appear in the symbol table in order to preserve scoping information. Therefore, the -v and -n options should be used only in conjunction with the -e option.

NAME

prof - display profile data

SYNOPSIS

prof [-tcan] [-ox] [-gl] [-z] [-h] [-s] [-m *mdata*] [*prog*]

DESCRIPTION

The prof command interprets a profile file produced by the monitor(3C) function. The symbol table in the object file *prog* (a.out by default) is read and correlated with a profile file (mon.out by default). For each text symbol the percentage of time spent executing between the address of that symbol and the address of the next is printed, together with the number of times that function was called and the average number of milliseconds per call.

The mutually exclusive options t, c, a, and n determine the type of sorting of the output lines:

- -t Sort by decreasing percentage of total time (default).
- -c Sort by decreasing number of calls.
- -a Sort by increasing symbol address.
- -n Sort lexically by symbol name.

The mutually exclusive options o and x specify the format (or base) for printing the address of each symbol monitored:

- -o Print each symbol address (in octal) along with the symbol name.
- -x Print each symbol address (in hexadecimal) along with the symbol name.

The mutually exclusive options -g and -l control the type of symbols to be reported. The -l option must be used with care; it applies the time spent in a static function to the preceding (in memory) global function, instead of giving the static function a separate entry in the report. If all static functions are properly located (see example below), this feature can be very useful. If not, the resulting report may be misleading.

Assume that A and B are global functions and only A calls static function S. If S is located immediately after A in the source code (that is, if S is properly located), then, with the -l option, the amount of time spent in A can easily be determined, including the time spent in S. If, however, both A and B call S, then, if the -l option is used, the report will be misleading; the time spent during B's call to S will be attributed to A, making it appear as if more time had been spent in A than really had. In this case, function S cannot be properly located.

- -g Include static (non-global) functions.
- -l Do not include static (non-global) functions (default).

The following options may be used in any combination:

- -z Include all symbols in the profile range [see monitor(3C)], even if associated with zero number of calls and zero time.
- -h Suppress the heading normally printed on the report. (This is useful if the report is to be processed further.)
- -s Print a summary of several of the monitoring parameters and statistics on the standard error output.
- -m *mdata*
  Use file *mdata* instead of mon.out as the input profile file.

-v Print `prof` version information on the standard error output.

A program creates a profile file if has been compiled with the -p option of cc(1). This option to the cc command arranges for calls to `monitor`(3C) at the beginning and end of execution. It is the call to `monitor` at the end of execution that causes a profile file to be written. The number of calls to a function is tallied if the -p option was used to compile the file containing the function.

The name of the file created by a profiled program is controlled by the environment variable PROFDIR. If PROFDIR does not exist, the file `mon.out` is produced in the current directory. If PROFDIR = *string*, the file *string/pid.progname* is produced, where *progname* consists of argv[0] with any path prefix removed, and *pid* is the program's process id. If PROFDIR is the null string, no profiling output is produced.

A single function may be split into subfunctions for profiling by means of the MARK macro [see `prof`(5)].

FILES

`mon.out` default profile file
`a.out` default namelist (object) file

SEE ALSO

cc(1), `exit`(2), `profil`(2), `monitor`(3C), `prof`(5).

NOTES

The times reported in successive identical runs may show variances because of varying cache-hit ratios that result from sharing the cache with other processes. Even if a program seems to be the only one using the machine, hidden background or asynchronous processes may blur the data. In rare cases, the clock ticks initiating recording of the program counter may "beat" with loops in a program, grossly distorting measurements. Call counts are always recorded precisely, however.

Only programs that call `exit` or return from `main` are guaranteed to produce a profile file, unless a final call to `monitor` is explicitly coded.

The times for static functions are attributed to the preceding external text symbol if the -g option is not used. However, the call counts for the preceding function are still correct; that is, the static function call counts are not added to the call counts of the external function.

If more than one of the options -t, -c, -a, and -n is specified, the last option specified is used and the user is warned.

Profiling may be used with dynamically linked executables, but care must be applied. Currently, shared objects cannot be profiled with `prof`. Thus, when a profiled, dynamically linked program is executed, only the "main" portion of the image is sampled. This means that all time spent outside of the "main" object, that is, time spent in a shared object, will not be included in the profile summary; the total time reported for the program may be less than the total time used by the program.

Because the time spent in a shared object cannot be accounted for, the use of shared objects should be minimized whenever a program is profiled with `prof`. If possible, the program should be linked statically before being profiled.

Consider an extreme case. A profiled program dynamically linked with the shared C library spends 100 units of time in some `libc` routine, say, `malloc`. Suppose `malloc` is called only from routine `B` and `B` consumes only 1 unit of time. Suppose further that routine `A` consumes 10 units of time, more than any other routine in the "main" (profiled) portion of the image. In this case, `prof` will conclude that most of the time is being spent in `A` and almost no time is being spent in `B`. From this it will be almost impossible to tell that the greatest improvement can be made by looking at routine B and not routine A. The value of the profiler in this case is severely degraded; the solution is to use archives as much as possible for profiling.

NAME

prs – print an SCCS file

SYNOPSIS

prs [-d[*dataspec*]] [-r[*SID*]] [-e] [-l] [-c[*date-time*]] [-a] *files*

DESCRIPTION

Prs prints, on the standard output, parts or all of an SCCS file (see sccsfile(4)) in a user-supplied format. If a directory is named, prs treats each file in the directory 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 or directory to be processed; non-SCCS files and unreadable files are silently ignored.

Arguments to prs, which may appear in any order, consist of *keyletter* arguments, and file names.

All the described *keyletter* arguments apply independently to each named file:

-d[*dataspec*] Used to specify the output data specification. The *dataspec* is a string consisting of SCCS file *data keywords* (see **Data Keywords**) interspersed with optional, user-supplied text.

-r[*SID*] Specifies the SCCSIDentification (SID) string of a delta for which information is desired. If no SID is specified, the SID of the most recently created delta is assumed. information for all deltas created *earlier* than and including the delta designated via the -r keyletter or the date given by the -c option. information for all deltas created *later* than and including the delta designated via the -r keyletter or the date given by the -c option. Cutoff date-time, in the form:

YY[MM[DD[HH[MM[SS]]]]]

-c[*date-time*] 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 in the form: "-c77/2/2 9:22:25". printing of information for both removed, i.e., delta type = *R*, (see rmdel(1)) and existing, i.e., delta type = *D*, deltas. If the -a keyletter is not specified, information for existing deltas only is provided.

Data Keywords

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file (see sccsfile(4)) have an associated data keyword. A data keyword may appear in a *dataspec* any number of times.

Prs prints: (1) the user-supplied text; and (2) appropriate values (extracted from the SCCS file) substituted for the recognized data keywords as they appear in the *dataspec*. The format of a data keyword value is either *Simple* SCCS, in which keyword substitution is direct, or *Multi-line* , in which keyword substitution is followed by a carriage return.

User-supplied text is any text other than recognized data keywords.
A tab is specified by \t and carriage return/new-line is specified by \n. The default data keywords are:

":Dt:\t:DL:\nMRs:\n:MR:COMMENTS:\n:C:"

TABLE 1. SCCS Files Data Keywords

Keyword	*Data Item*	*File Section*	*Value*	*Format*
:Dt:	Delta information	Delta Table	See below*	S
:DL:	Delta line statistics	"	:Li:/:Ld:/:Lu:	S
:Li:	Lines inserted by delta	"	*nnnnn*	S
:Ld:	Lines deleted by delta	"	*nnnnn*	S
:Lu:	Lines unchanged by delta	"	*nnnnn*	S
:DT:	Delta type	"	D or R	S
:I:	SCCS ID string (SID)	"	:R:.:L:.:B:.:S:	S
:R:	Release number	"	*nnnn*	S
:L:	Level number	"	*nnnn*	S
:B:	Branch number	"	*nnnn*	S
:S:	Sequence number	"	*nnnn*	S
:D:	Date delta created	"	:Dy:/:Dm:/:Dd:	S
:Dy:	Year delta created	"	*nn*	S
:Dm:	Month delta created	"	*nn*	S
:Dd:	Day delta created	"	*nn*	S
:T:	Time delta created	"	:Th:::Tm:::Ts:	S
:Th:	Hour delta created	"	*nn*	S
:Tm:	Minutes delta created	"	*nn*	S
:Ts:	Seconds delta created	"	*nn*	S
:P:	Programmer who created delta	"	logname	S
:DS:	Delta sequence number	"	*nnnn*	S
:DP:	Predecessor delta seq-no.	"	*nnnn*	S
:DI:	Seq-no. of deltas incl., excl., ignored	"	:Dn:/:Dx:/:Dg:	S
:Dn:	Deltas included (seq #)	"	:DS: :DS:...	S
:Dx:	Deltas excluded (seq #)	"	:DS: :DS:...	S
:Dg:	Deltas ignored (seq #)	"	:DS: :DS:...	S
:MR:	MR numbers for delta	"	text	M
:C:	Comments for delta	"	text	M
:UN:	User names	User Names	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag	"	text	S
:MF:	MR validation flag	"	yes or no	S
:MP:	MR validation pgm name	"	text	S
:KF:	Keyword error/warning flag	"	yes or no	S
:KV:	Keyword validation string	"	text	S
:BF:	Branch flag	"	yes or no	S
:J:	Joint edit flag	"	yes or no	S
:LK:	Locked releases	"	:R: . . .	S
:Q:	User defined keyword	"	text	S
:M:	Module name	"	text	S
:FB:	Floor boundary	"	:R:	S
:CB:	Ceiling boundary	"	:R:	S
:Ds:	Default SID	"	:I:	S
:ND:	Null delta flag	"	yes or no	S
:FD:	File descriptive text	Comments	text	M
:BD:	Body	Body	text	M
:GB:	Gotten body	"	text	M
:W:	A form of what(1) string	N/A	:Z::M:\t:I:	S
:A:	A form of what(1) string	N/A	:Z::Y: :M: :I::Z:	S
:Z:	what(1) string delimiter	N/A	@(#)	S

:F:	SCCS file name	N/A	text	S
:PN:	SCCS file path name	N/A	text	S

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

EXAMPLES

```
prs -d"Users and/or user IDs for :F: are:\n:UN:" s.file
```

may produce on the standard output:

```
Users and/or user IDs for s.file are:
    xyz
    131
    abc
```

```
prs -d"Newest delta for pgm :M:: :I: Created :D: By :P:" -r s.file
```

may produce on the standard output:

```
Newest delta for pgm main.c: 3.7 Created 77/12/1 By cas
```

As a *special case:*

```
prs s.file
```

may produce on the standard output:

```
D 1.1 77/12/1 00:00:00 cas 1 000000/00000/00000
MRs:
bl78-12345
bl79-54321
COMMENTS:
this is the comment line for s.file initial delta
```

for each delta table entry of the D type. The only keyletter argument allowed to be used with the *special case* is the -a keyletter.

FILES

/tmp/pr?????

DIAGNOSTICS

Use help(1) for explanations.

SEE ALSO

admin(1), delta(1), get(1), help(1).
sccsfile(4) in the *Programmer's Reference for the DG/UX System*
"Source Code Control System" in *Programmer's Guide: ANSI C and Programming Support Tools*.

NAME

ratfor – rational FORTRAN dialect

SYNOPSIS

ratfor [*options*] [*files*]

DESCRIPTION

Ratfor converts a rational dialect of FORTRAN into ordinary irrational FORTRAN. Ratfor provides control flow constructs essentially identical to those in C:

statement grouping:

```
{ statement; statement; statement }
```

decision-making:

```
if (condition) statement [ else statement ]
switch (integer value) {
        case integer: statement
        ...
        [ default: ] statement
}
```

loops:

```
while (condition) statement
for (expression; condition; expression) statement
do limits statement
repeat statement [ until (condition) ]
break
next
```

and some syntactic sugar to make programs easier to read and write:

free form input:

multiple statements/line; automatic continuation

comments:

this is a comment.

translation of relationals:

>, >=, etc., become .GT., .GE., etc.

return expression to caller from function:

return (*expression*)

define:

define *name replacement*

include:

include *file*

The option -h causes quoted strings to be turned into 27H constructs. The -C option copies comments to the output and attempts to format it neatly. Normally, continuation lines are marked with a & in column 1; the option -6x makes the continuation character x and places it in column 6.

Ratfor is best used with f77(1).

SEE ALSO

f77(1).

B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.

NAME

rcs – change RCS file attributes

SYNOPSIS

rcs [*options*] *file* ...

DESCRIPTION

Rcs creates new RCS files or changes attributes of existing ones. An RCS file contains multiple revisions of text, an access list, a change log, descriptive text, and some control attributes. For rcs to work, the caller's login name must be on the access list, except if the access list is empty, the caller is the owner of the file or the superuser, or the -i option is present.

Files ending in ',v' are RCS files, all others are working files. If a working file is given, rcs tries to find the corresponding RCS file first in directory ./RCS and then in the current directory, as explained in co(1).

-i
: creates and initializes a new RCS file, but does not deposit any revision. If the RCS file has no path prefix, rcs tries to place it first into the subdirectory ./RCS, and then into the current directory. If the RCS file already exists, an error message is printed.

-a*logins*
: appends the names appearing in the comma-separated list *logins* to the access list of the RCS file.

-A*oldfile*
: appends the access list of *oldfile* to the access list of the RCS file.

-e[*logins*]
: erases the login names appearing in the comma-separated list *logins* from the access list of the RCS file. If *logins* is omitted, the entire access list is erased.

-c*string*
: sets the comment leader to *string*. The comment leader is printed before every log message line generated by the keyword Log during checkout (see co). This is useful for programming languages without multi-line comments. During rcs -i or initial ci, the comment leader is guessed from the suffix of the working file.

-l[*rev*]
: locks the revision with number *rev*. If a branch is given, the latest revision on that branch is locked. If *rev* is omitted, the latest revision on the trunk is locked. Locking prevents overlapping changes. A lock is removed with ci or rcs -u (see below).

-u[*rev*]
: unlocks the revision with number *rev*. If a branch is given, the latest revision on that branch is unlocked. If *rev* is omitted, the latest lock held by the caller is removed. Normally, only the locker of a revision may unlock it. Somebody else unlocking a revision breaks the lock. This causes a mail message to be sent to the original locker. The message contains a commentary solicited from the breaker. The commentary is terminated with a line containing a single '.' or control-D.

-L
: sets locking to strict. Strict locking means that the owner of an RCS file is not exempt from locking for checkin. This option should be used for files that are shared.

-U
: sets locking to non-strict. Non-strict locking means that the owner of a file need not lock a revision for checkin. This option should NOT be used for files that are shared. The default is -L.

-n*name*[:*rev*]
: associates the symbolic name *name* with the branch or revision *rev*. Rcs

prints an error message if *name* is already associated with another number. If *rev* is omitted, the symbolic name is deleted. Names must begin with a letter, and cannot contain whitespace, period, colon, semicolon, or @.

-N*name*[:*rev*]
: same as -n, except that it overrides a previous assignment of *name*.

-o*range*
: deletes ("outdates") the revisions given by *range*. A range consisting of a single revision number means that revision. A range consisting of a branch number means the latest revision on that branch. A range of the form *rev1*–*rev2* means revisions *rev1* to *rev2* on the same branch, –*rev* means from the beginning of the branch containing *rev* up to and including *rev*, and *rev*– means from revision *rev* to the end of the branch containing *rev*. None of the outdated revisions may have branches or locks.

-q
: quiet mode; diagnostics are not printed.

-s*state*[:*rev*]
: sets the state attribute of the revision *rev* to *state*. If *rev* is omitted, the latest revision on the trunk is assumed; If *rev* is a branch number, the latest revision on that branch is assumed. Any string that could be a name (see -n) is acceptable for *state*. A useful set of states is `Exp` (for experimental), `Stab` (for stable), and `Rel` (for released). By default, `ci` sets the state of a revision to `Exp`.

-t[*txtfile*]
: writes descriptive text into the RCS file (deletes the existing text). If *txtfile* is omitted, `rcs` prompts the user for text supplied from the std. input, terminated with a line containing a single '.' or control-D. Otherwise, the descriptive text is copied from the file *txtfile*. If the -i option is present, descriptive text is requested even if -t is not given. The prompt is suppressed if standard input is not a terminal.

FILES

The caller of the command must have read/write permission for the directory containing the RCS file and read permission for the RCS file itself. `Rcs` creates a semaphore file in the same directory as the RCS file to prevent simultaneous update. For changes, `rcs` always creates a new file. On successful completion, `rcs` deletes the old one and renames the new one. This strategy makes links to RCS files useless.

DIAGNOSTICS

The RCS file name and the revisions outdated are written to the diagnostic output. The exit status always refers to the last RCS file operated upon, and is 0 if the operation was successful, 1 otherwise.

SEE ALSO

`co`(1), `ci`(1), `ident`(1), `rcsdiff`(1), `rcsintro`(1), `rcsmerge`(1), `rlog`(1), `sccstorcs`(1), `rcsfile`(4).

Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

rcsdiff - compare RCS revisions

SYNOPSIS

rcsdiff [**-biwt**] [**-cefhn**] [**-r***rev1*] [**-r***rev2*] *file* ...

DESCRIPTION

Rcsdiff runs **berk_diff**(1) to compare two revisions of each RCS file given. A file name ending in ,v is an RCS file name, otherwise a working file name. **Rcsdiff** derives the working file name from the RCS file name and vice versa, as explained in co(1). Pairs consisting of both an RCS and a working file name may also be specified.

All options except **-r** have the same effect as described in **berk_diff**(1).

If both *rev1* and *rev2* are omitted, **rcsdiff** compares the latest revision on the trunk with the contents of the corresponding working file. This is useful for determining what you changed since the last checkin.

If *rev1* is given, but *rev2* is omitted, **rcsdiff** compares revision *rev1* of the RCS file with the contents of the corresponding working file.

If both *rev1* and *rev2* are given, **rcsdiff** compares revisions *rev1* and *rev2* of the RCS file.

Both *rev1* and *rev2* may be given numerically or symbolically.

The environment variable RCS_DIFF controls what *diff* program rcsdiff will run. You can set RCS_DIFF to diff(1) or to your own alternate *diff* program. If this environment variable is not set, berk_diff(1) will be run.

EXAMPLES

The command

```
rcsdiff  f.c
```

runs berk_diff on the latest trunk revision of RCS file f.c,v and the contents of working file f.c.

SEE ALSO

ci(1), co(1), berk_diff(1), ident(1), rcs(1), rcsintro(1), rcsmerge(1), rlog(1), rcsfile(4).

Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

`rcsintro` – introduction to RCS commands

DESCRIPTION

The Revision Control System (RCS) manages multiple revisions of text files. RCS automates the storing, retrieval, logging, identification, and merging of revisions. RCS is useful for text that is revised frequently, for example programs, documentation, graphics, papers, form letters, etc.

The basic user interface is extremely simple. The novice only needs to learn two commands: `ci` and `co`. `Ci`, short for "check in", deposits the contents of a text file into an archival file called an RCS file. An RCS file contains all revisions of a particular text file. `Co`, short for "check out", retrieves revisions from an RCS file.

SEE ALSO

`ci`(1), `co`(1), `ident`(1), `merge`(1), `rcs`(1), `rcsdiff`(1), `rcsmerge`(1), `rlog`(1), `rcsfile`(4).
Walter F. Tichy, "An Introduction to the Revision Control System", Programmer Supplementary Documents, Volume 1 (PS1), #13 (This document is available from the *University of California Berkeley*)

NAME

`rcsmerge` – merge RCS revisions

SYNOPSIS

`rcsmerge` `-r`*rev1* [`-r`*rev2*] [`-p`] *file*

DESCRIPTION

`Rcsmerge` incorporates the changes between *rev1* and *rev2* of an RCS file into the corresponding working file. If `-p` is given, the result is printed on the std. output, otherwise the result overwrites the working file.

A file name ending in ',v' is an RCS file name, otherwise a working file name. `Merge` derives the working file name from the RCS file name and vice versa, as explained in co(1). A pair consisting of both an RCS and a working file name may also be specified.

Rev1 may not be omitted. If *rev2* is omitted, the latest revision on the trunk is assumed. Both *rev1* and *rev2* may be given numerically or symbolically.

`Rcsmerge` prints a warning if there are overlaps, and delimits the overlapping regions as explained in `co -j`. The command is useful for incorporating changes into a checked-out revision.

EXAMPLES

Suppose you have released revision 2.8 of f.c. Assume furthermore that you just completed revision 3.4, when you receive updates to release 2.8 from someone else. To combine the updates to 2.8 and your changes between 2.8 and 3.4, put the updates to 2.8 into file f.c and execute

```
rcsmerge  -p  -r2.8  -r3.4  f.c  >f.merged.c
```

Then examine f.merged.c. Alternatively, if you want to save the updates to 2.8 in the RCS file, check them in as revision 2.8.1.1 and execute `co -j`:

```
ci  -r2.8.1.1  f.c
co  -r3.4  -j2.8:2.8.1.1  f.c
```

As another example, the following command undoes the changes between revision 2.4 and 2.8 in your currently checked out revision in f.c.

```
rcsmerge  -r2.8  -r2.4  f.c
```

Note the order of the arguments, and that f.c will be overwritten.

SEE ALSO

`ci`(1), `co`(1), `merge`(1), `ident`(1), `rcs`(1), `rcsdiff`(1), `rlog`(1), `rcsfile`(4).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NOTES

`Rcsmerge` does not work for files that contain lines with a single '.'.

NAME

regcmp – regular expression compile

SYNOPSIS

regcmp [–] *file*...

DESCRIPTION

The regcmp command performs a function similar to regcmp and, in most cases, precludes the need for calling regcmp from C programs. Bypassing regcmp saves on both execution time and program size. The command regcmp compiles the regular expressions in *file* and places the output in *file*.i. If the – option is used, the output is placed in *file*.c. The format of entries in *file* is a name (C variable) followed by one or more blanks followed by one or more regular expressions enclosed in double quotes. The output of regcmp is C source code. Compiled regular expressions are represented as extern char vectors. *file*.i files may thus be #included in C programs, or *file*.c files may be compiled and later loaded. In the C program that uses the regcmp output, regex(abc,line) applies the regular expression named abc to line. Diagnostics are self-explanatory.

EXAMPLES

```
name     "([A-Za-z][A-Za-z0-9_]*)$0"

telno    "\({0,1}([2-9][01][1-9])$0\){0,1} *"
         "([2-9][0-9]{2})$1[ -]{0,1}"
         "([0-9]{4})$2"
```

The three arguments to telno shown above must all be entered on one line.

In the C program that uses the regcmp output,

```
regex(telno, line, area, exch, rest)
```

applies the regular expression named telno to line.

SEE ALSO

regcmp(3G).

NOTES

The regcmp command and the application code that calls the regex routine with the compiled regular expression must be run in the same locale. See setlocale(3C).

NAME

rev – reverse order of characters in each line of file

SYNOPSIS

rev [*file* ...]

where:

file Name of input file; if no file is specified, standard input is used.

DESCRIPTION

Rev copies the named files to the standard output, reversing the order of characters in every line.

SEE ALSO

awk(1), dd(1).

NAME

`rlog` – print log messages and other information about RCS files

SYNOPSIS

`rlog` [*options*] *file* ...

DESCRIPTION

`Rlog` prints information about RCS files. Files ending in ',v' are RCS files, all others are working files. If a working file is given, `rlog` tries to find the corresponding RCS file first in directory ./RCS and then in the current directory, as explained in `co`(1).

`Rlog` prints the following information for each RCS file: RCS file name, working file name, head (i.e., the number of the latest revision on the trunk), access list, locks, symbolic names, suffix, total number of revisions, number of revisions selected for printing, and descriptive text. This is followed by entries for the selected revisions in reverse chronological order for each branch. For each revision, `rlog` prints revision number, author, date/time, state, number of lines added/deleted (with respect to the previous revision), locker of the revision (if any), and log message. Without options, `rlog` prints complete information. The options below restrict this output.

`-L` ignores RCS files that have no locks set; convenient in combination with `-R`, `-h`, or `-l`.

`-R` only prints the name of the RCS file; convenient for translating a working file name into an RCS file name.

`-h` prints only RCS file name, working file name, head, access list, locks, symbolic names, and suffix.

`-t` prints the same as `-h`, plus the descriptive text.

`-d`*dates* prints information about revisions with a checkin date/time in the ranges given by the semicolon-separated list of *dates*. A range of the form *d1*<*d2* or *d2*>*d1* selects the revisions that were deposited between *d1* and *d2*, (inclusive). A range of the form <*d* or *d*> selects all revisions dated *d* or earlier. A range of the form *d*< or >*d* selects all revisions dated *d* or later. A range of the form *d* selects the single, latest revision dated *d* or earlier. The date/time strings *d, d1,* and *d2* are in the free format explained in `co`(1). Quoting is normally necessary, especially for < and >. Note that the separator is a semicolon.

`-l`[*lockers*]
prints information about locked revisions. If the comma-separated list *lockers* of login names is given, only the revisions locked by the given login names are printed. If the list is omitted, all locked revisions are printed.

`-r`*revisions*
prints information about revisions given in the comma-separated list *revisions* of revisions and ranges. A range *rev1*–*rev2* means revisions *rev1* to *rev2* on the same branch, –*rev* means revisions from the beginning of the branch up to and including *rev*, and *rev*– means revisions starting with *rev* to the end of the branch containing *rev*. An argument that is a branch means all revisions on that branch. A range of branches means all revisions on the branches in that range.

`-s`*states* prints information about revisions whose state attributes match one of the states given in the comma-separated list *states*.

−w[*logins*] prints information about revisions checked in by users with login names appearing in the comma-separated list *logins*. If *logins* is omitted, the user's login is assumed.

Rlog prints the intersection of the revisions selected with the options −d, −l, −s, −w, and −r.

EXAMPLES

```
rlog -L -R RCS/*,v
rlog -L -h RCS/*,v
rlog -L -l RCS/*,v
rlog RCS/*,v
```

The first command prints the names of all RCS files in the subdirectory 'RCS' which have locks. The second command prints the headers of those files, and the third prints the headers plus the log messages of the locked revisions. The last command prints complete information.

DIAGNOSTICS

The exit status always refers to the last RCS file operated upon, and is 0 if the operation was successful, 1 otherwise.

SEE ALSO

ci(1), co(1), ident(1), rcs(1), rcsdiff(1), rcsintro(1), rcsmerge(1), rcsfile(4), sccstorcs(8).

Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

rmdel – remove a delta from an SCCS file

SYNOPSIS

rmdel -r *SID files*

DESCRIPTION

Rmdel removes the delta specified by the *SID* from each named SCCS file. The delta to be removed must be the most recent delta in its branch in the delta chain of each named SCCS file. In addition, the *SID* specified must *not* be that of a version being edited for the purpose of making a delta: if a *p-file* (see get(1)) exists for the named SCCS file, the *SID* specified must *not* appear in any entry of the *p-file*

If a directory is named, rmdel treats each file in the directory 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; non-SCCS files and unreadable files are silently ignored.

Simply stated, if you make a delta you can remove it; or if you own the file and directory you can remove a delta.

EXAMPLES

```
$ rmdel -r1.15 /work/archives/s.file1
```

This command specifies the removal of delta '1.15' of the SCCS file 's.file1'. This process will cause the delta's type indicator in the "delta table" of the SCCS file to be changed from "D" (delta) to "R" (removed).

```
$ rmdel -r1.3.1.1 s.file2
```

This command specifies the removal of branch delta '1.3.1.1' from the SCCS file 's.file2'. This will also cause the delta's type indicator to change from "D" to "R".

FILES

x.file [see delta(1)]
z.file [see delta(1)]

DIAGNOSTICS

Use help(1) for explanations.

SEE ALSO

delta(1), get(1), help(1), prs(1).
sccsfile(4) in the *Programmer's Reference for the DG/UX System*
"Source Code Control System" in *Programmer's Guide: ANSI C and Programming Support Tools*.

NAME

`sccsdiff` – compare two versions of an SCCS file

SYNOPSIS

`sccsdiff -r`*SID1* `-r`*SID2* [-p] [-*sn*] *files*

DESCRIPTION

`Sccsdiff` compares two versions of an SCCS file and generates the differences between them. Any number of SCCS files may be specified, but arguments apply to all files.

`-r`*SID?* — *SID1* and *SID2* specify the deltas of an SCCS file that are to be compared. Versions are passed to `bdiff`(1) in the order given.

`-p` — Pipe output for each file through `pr`(1).

-*sn* — *N* is the file segment size that `bdiff` will pass to `diff`(1). This is useful when `diff` fails due to a high system load.

EXAMPLES

```
sccsdiff -r1.3 -r1.4 /work/archives/s.file1
```

This command lists differences (if there are any) between versions '1.3' and '1.4' of the SCCS file 's.file1'. If the versions are identical, you will get the message 's.file1: No differences'.

```
sccsdiff -r1.3 -r1.4 -p s.file2
```

This command does the same as the previous example, except the output is formatted.

```
sccsdiff -r1.5.1.1 -r1.5.1.2 -s100 s.file3
```

This command lists any differences between versions '1.5.1.1' and '1.5.1.2' of the SCCS file 's.file3'. The -s100 will pass 100 segments at a time from 'bdiff' to 'diff'. This is useful under high system load.

FILES

`/tmp/get?????` Temporary files

DIAGNOSTICS

You get the message

```
file: No differences
```

if the two versions are the same.

Use `help`(1) for explanations.

SEE ALSO

`bdiff`(1), `get`(1), `help`(1), `pr`(1).
"Source Code Control System" in *Programmer's Guide: ANSI C and Programming Support Tools*

NAME

sccstorcs – build RCS file from SCCS file

SYNOPSIS

sccstorcs [-t] [-v] [-c*shell-cmd*] *s.file* ...

DESCRIPTION

Sccstorcs builds an RCS file from each SCCS file argument. The deltas and comments for each delta are preserved and installed into the new RCS file in order. Also preserved are the user access list and descriptive text, if any, from the SCCS file.

The following flags are meaningful:

-c*shell-cmd*
: Executes *shell-cmd* for each revision before installing it in the RCS file. Occurrences of %s in *shell-cmd* are replaced by the name of the file containing the revision.

-t
: Trace only. Prints detailed information about the SCCS file and lists the commands that would be executed to produce the RCS file. No commands are actually executed and no RCS file is made.

-v
: Verbose. Prints each command that is run while it is building the RCS file.

FILES

For each s.*somefile*, Sccstorcs writes the files *somefile* and *somefile*,v which should not already exist. Sccstorcs will abort, rather than overwrite those files if they do exist.

DIAGNOSTICS

All diagnostics are written to stderr. Errors cause a non-zero exit status.

SEE ALSO

ci(1), co(1), rcs(1).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

Sccstorcs does not preserve all SCCS options specified in the SCCS file. Most notably, it does not preserve removed deltas, MR numbers, and cutoff points.

NAME

sdb – symbolic debugger

SYNOPSIS

sdb [–s*signo*] [–V] [–W] [–w] [*objfile* [*corfile* [*directory-list*]]]

DESCRIPTION

sdb is the symbolic debugger for C, F77, and assembly programs. sdb may be used to examine executable program files and core files. It may also be used to examine live processes in a controlled execution environment.

The *objfile* argument is the name of an executable program file. To take full advantage of the symbolic capabilities of sdb, this file should be compiled with the –g (debug) option. If it has not been compiled with the –g option, the symbolic capabilities of sdb will be limited, but the file can still be examined and the program debugged.

The *corfile* argument is the name of a core image file. A core image file is produced by the abnormal termination of *objfile*. The default for *corfile* is core. A core image file need not be present to use sdb. Using a hyphen (–) instead of *corfile* forces sdb to ignore an existing core image file.

The *directory-list* argument is a colon-separated list of directories that is used by sdb to locate source files used to build *objfile*. If no directory list is specified, sdb will look in the current directory.

The following options are recognized by sdb:

–s *signo*
: Where *signo* is a decimal number that corresponds to a signal number [see signal(2)], do not stop live processes under control of sdb that receive the signal. This option may be used more than once on the sdb command line.

–V
: Print version information. If no *objfile* argument is specified on the command line, sdb will exit after printing the version information.

–W
: Suppress warnings about *corfile* being older than *objfile* or about source files that are older than *objfile*.

–w
: Allow user to write to *objfile* or *corfile* (this option is supported only when debugging COFF files).

sdb recognizes a current line and a current file. When sdb is examining an executable program file without a core file, the current line and current file are initially set to the line and file containing the first line of main. If *corfile* exists, then current line and current file are initially set to the line and file containing the source statement where the process terminated. Note that on the 88K, this may not be the instruction which actually caused the process to terminate. The current line and current file change automatically as a live process executes. They may also be changed with the source file examination commands.

Names of variables are written as in C. Variables local to a procedure may be accessed using the form *procedure*:*variable*. If no procedure name is given, the procedure containing the current line is used by default.

Structure members may be referred to as *variable*.*member*, pointers to structure members as *variable*–>*member*, and array elements as *variable*[*number*]. Pointers may also be dereferenced by using the form *pointer*[*number*]. Combinations of these forms may also be used. The form *number*–>*member* may be used where *number* is the address of a pointer, and *number*.*member* where *number* is interpreted as the address of a structure instance. The template of the structure type used in this case will be the last structure type referenced. When sdb displays the value of a structure, it does so by displaying the value of all elements of the structure. The address of a structure is displayed by displaying the address of the structure instance rather than the addresses of individual elements.

Elements of a multidimensional array may be referred to as *variable* [*number*] [*number*] ..., or as *variable* [*number*, *number*, ...]. In place of *number*, the form *number*; *number* may be used to indicate a range of values, * may be used to indicate all legitimate values for that subscript, or subscripts may be omitted entirely if they are the last subscripts and the full range of values is desired. If no subscripts are specified, sdb will display the value of all elements of the array.

A particular instance of a variable on the stack is referred to as *procedure*:*variable*, *number*. The *number* is the occurrence of the specified procedure on the stack, with the topmost occurrence being 1. The default procedure is the one containing the current line.

Addresses may be used in sdb commands as well. Addresses are specified by decimal, octal, or hexadecimal numbers.

Line numbers in the source program are specified by the form *filename*:*number* or *procedure*:*number*. In either case, the *number* is relative to the beginning of the file and corresponds to the line number used by text editors or the output of pr. A number used by itself implies a line in the current file.

While a live process is running under sdb, all addresses and identifiers refer to the live process. When sdb is not examining a live process, the addresses and identifiers refer to *objfile* or *corfile*.

Commands

The commands for examining data in the program are:

t
: Prints a stack trace of the terminated or halted program. The function invoked most recently is at the top of the stack. For C programs, the Stack ends with _start, which is the startup routine that invokes main.

T
: Prints the top line of the stack trace.

variable/*clm*
: Prints the value of *variable* according to length *l* and format *m*. The numeric count *c* indicates that a region of memory, beginning at the address implied by *variable*, is to be displayed. The length specifiers are:

  b one byte

  h two bytes (half word)

  l four bytes (long word)

  Legal values for *m* are:

  c character

  d signed decimal

  u unsigned decimal

  o octal

  x hexadecimal

X	hexadecimal (uppercase)

f	32-bit single precision floating point

g	64-bit double precision floating point

s	Assumes that *variable* is a string pointer and prints characters starting at the address pointed to by the variable.

a	Prints characters starting at the variable's address. Do not use this with register variables.

p	pointer to procedure

i	Disassembles machine-language instruction with addresses printed numerically and symbolically.

I	Disassembles machine-language instruction with addresses printed numerically only.

Length specifiers are effective with formats c, d, u, o, x. The length specifier determines the output length of the value to be displayed. This value may be truncated. The count specifier *c* displays that many units of memory, starting at the address of the *variable*. The number of bytes in the unit of memory is determined by *l* or by the size associated with the variable. If the specifiers *c*, *l*, and *m* are omitted, sdb uses defaults. If a count specifier is used with the s or a command, then that many characters are printed. Otherwise, successive characters are printed until either a null byte is reached or 128 characters are printed. The last variable may be redisplayed with the ./ command.

For a limited form of pattern matching, use the sh metacharacters * and ? within procedure and variable names. (sdb does not accept these metacharacters in file names, as the function name in a line number when setting a breakpoint, in the function call command, or as the argument to the e command.) If no procedure name is supplied, sdb matches both local and global variables. If the procedure name is specified, then sdb matches only local variables. To match global variables only, use :*pattern*. To print all variables, use *:*.

linenumber?*lm*
variable:?*lm*

Prints the value at the address from the executable or text space given by *linenumber* or *variable* (procedure name), according to the format *lm*. The default format is i.

variable=*lm*
linenumber=*lm*
number=*lm*

Prints the address of *variable* or *linenumber*, or the value of *number*. *l* specifies length and *m* specifies the format. If no format is specified, then sdb uses lx (four-byte hex). *m* allows you to convert between decimal, octal, and hexadecimal.

variable!*value*

Sets *variable* to the given *value*. The value may be a number, a character constant, or a variable. The value must be well-defined; structures are allowed only if assigning to another structure variable of the same type. Character constants are denoted ^*character*. Numbers are viewed as integers unless a decimal point or exponent is used. In this case, they are treated as having the type double.

Registers, except the floating point registers, are viewed as integers. Register

31, as well as the special-function register names (such as `fp` and names are `r0-`
`sp`) recognized by the assembler. `sdb` recognizes register names by a prepended or appended `%`, as in `%r6` or `fp%`. When debugging a COFF object, only the form with appended `%` is accepted.

If the address of a variable is given, it is regarded as the address of a variable of type `int`. C conventions are used in any type conversions necessary to perform the indicated assignment. If `sdb` is invoked with the `-w` flag, writing to text addresses before the execution of the program, or after its completion, will change the actual values in the objfile. Writing to these addresses during program execution will change only the image in memory.

`x` Prints the machine registers and the current machine-language instruction.

`X` Prints the current machine-language instruction.

The commands for examining source files are:

e
e *procedure*
e *filename*
e *directory/*

e, without arguments, prints the name of the current file. The second form sets the current file to the file containing the procedure. The third form sets the current file to *filename*. The current line is set to the first line in the named procedure or file. Source files are assumed to be in the directories in the directory list. The fourth form adds *directory* to the end of the directory list.

/regular expression/

Searches forward from the current line for a line containing a string matching *regular expression*, as in `ed`. The trailing / may be omitted, except when associated with a breakpoint.

?regular expression?

Searches backward from the current line for a line containing a string matching *regular expression*, as in `ed`. The trailing ? may be omitted, except when associated with a breakpoint.

`p` Prints the current line.

`z` Prints the current line and the following nine lines. Sets the current line to the last line printed.

`w` Prints the 10 lines (the window) around the current line.

number

Specifies the current line. Prints the new current line.

count+

Advances the current line by *count* lines. Prints the new current line.

count–

Resets the current line by *count* lines back. Prints the new current line.

The commands for controlling the execution of the source program are:

count `r` *args*
count `R`

Runs the program with the given arguments. The `r` command with no arguments reuses the previous arguments to the program. The `R` command runs the program with no arguments. An argument beginning with < or > redirects the standard input or output, respectively. Full sh syntax is accepted. If *count* is given, sdb stops when it encounters *count* breakpoints.

linenumber c *count*
linenumber C *count*
: Continues execution. sdb stops when it encounters *count* breakpoints. The signal that stopped the program is reactivated with the C command and ignored with the c command. If a line number is specified, then a temporary breakpoint is placed at the line and execution continues. The breakpoint is deleted when the command finishes.

linenumber g *count*
: Continues with execution resumed at the given line. If *count* is given, sdb stops when it encounters *count* breakpoints. Results are undefined if *linenumber* is in a different context (e.g. another procedure).

s *count*
S *count*
: s single steps the program through *count* lines; or if no *count* is given, the program runs for one line. s will step from one function into a called function. S also steps a program, but it will not step into a called function. It steps over the function called.

i *count*
I *count*
: Single steps by *count* machine-language instructions. The signal that caused the program to stop is reactivated with the I command and ignored with the i command.

variable$m *count*
address:m *count*
: Single steps (as with s) until the specified location is modified with a new value. If *count* is omitted, it is, in effect, infinity. *Variable* must be accessible from the current procedure. This command can be very slow.

level v
: Toggles verbose mode. This is for use when single stepping with S, s, or m. If *level* is omitted, then just the current source file and/or function name is printed when either changes. If *level* is 1 or greater, each C source line is printed before it executes. If *level* is 2 or greater, each assembler statement is also printed. A v turns verbose mode off.

k Kills the program being debugged.

procedure(*arg1*,*arg2*,...)
procedure(*arg1*,*arg2*,...)/*m*
: Executes the named procedure with the given arguments. Arguments can be register names, integer, character, or string constants, or names of variables accessible from the current procedure. The second form causes the value returned by the procedure to be printed according to format *m*. If no format is given, it defaults to d.

linenumber b *commands*
: Sets a breakpoint at the given line. If a procedure name without a line number is given (e.g., *proc*:), a breakpoint is placed at the first line in the procedure even if it was not compiled with the -g option. If no *linenumber* is given, a breakpoint is placed at the current line. If no *commands* are given, execution stops at the breakpoint and control is returned to sdb. Otherwise the

commands are executed when the breakpoint is encountered. Multiple commands are specified by separating them with semicolons. Nested associated commands are not permitted; setting breakpoints within the associated environments is permitted.

`B` Prints a list of the currently active breakpoints.

linenumber `d`
: Deletes a breakpoint at the given line. If no *linenumber* is given, then the breakpoints are deleted interactively. Each breakpoint location is printed and a line is read from the standard input. If the line begins with a `y` or `d`, then the breakpoint is deleted.

`D` Deletes all breakpoints.

`l` Prints the last executed line.

linenumber `a`
: Announces a line number. If *linenumber* is of the form *proc*`:`*number*, the command effectively does a *linenumber*`:b l;c`. If *linenumber* is of the form *proc*`:`, the command effectively does a *proc*`:b T;c`.

Miscellaneous commands:

#rest-of-line
: The *rest-of-line* represents comments that are ignored by `sdb`.

`!`*command*
: The *command* is interpreted by `sh`.

`new-line`
: If the previous command printed a source line, then advance the current line by one line and print the new current line. If the previous command displayed a memory location, then display the next memory location. If the previous command disassembled an instruction, then disassemble the next instruction.

`end-of-file character`
: Scrolls the next 10 lines of instructions, source, or data depending on which was printed last. The end-of-file character is usually `control-d`.

< *filename*
: Read commands from *filename* until the end of file is reached, and then continue to accept commands from standard input. Commands are echoed, preceded by two asterisks, just before being executed. This command may not be nested; < may not appear as a command in a file.

`M` Prints the address maps.

" *string* "
: Prints the given string. The C escape sequences of the form *character*, *octaldigits*, or \\x*hexdigits* are recognized, where *character* is a nonnumeric character. The trailing quote may be omitted.

`q` Exits the debugger.

`v` Prints version stamping information.

SEE ALSO

`cc`(1), `signal`(2), `a.out`(4), `core`(4), `syms`(4).
`ed`(1), `sh`(1) in the *User's Reference Manual*.
The "`sdb`" chapter in the *Programmer's Guide: ANSI C and Programming Support Tools*.

NOTES

When `sdb` prints the value of an external variable for which there is no debugging information, a warning is printed before the value. The size is assumed to be `int` (integer).

Data which are stored in text sections are indistinguishable from functions.

Line number information in optimized functions is unreliable, and some information may be missing.

Arguments in function calls are limited in size to 32 bits (pointers are allowed).

When debugging COFF executables, function calls from within `sdb` cannot be made before main is reached.

If *objfile* is a dynamically linked executable, variables, function names, and so on that are defined in shared objects may not be referenced until the shared object in which the variable, etc., is defined is attached to the process. For shared objects attached at startup (e.g. `libc.so.1`, the default C library), this implies that such variables may not be accessed until main is called.

The *objfile* argument is accessed directly for debugging information while the process is created via the PATH variable.

NAME

sde-target – print commands to reset software development environment target

SYNOPSIS

sde-target [-sh | -csh] [*target*]

DESCRIPTION

The sde-target command prints the shell command lines that you execute to reset your environment so that the software development tools produce code for a specified target (see sde(5)). The command lines reset your environment by setting the TARGET_BINARY_INTERFACE environment variable to the validated pathname component of a directory in /usr/sde.

The easiest way to use sde-target is to embed it in an eval command that you invoke via a C shell alias or a Bourne shell function (see sh(1) and csh(1)).

For example, the following csh(1) command creates an alias targ that invokes sde-target and executes the commands it returns:

```
alias targ 'eval `sde-target -csh \!*`'
```

The following sh(1) command creates a shell function that does the same:

```
targ () eval `sde-target -sh "$@"`
```

After you create targ or a similar alias or function, you can set your software development environment by invoking targ with the proper environment name.

OPTIONS

-sh Print commands in Bourne shell syntax.

-csh Print commands in C shell syntax.

target Specify the target system, for example m88kbcs. If you specify default, the environment is reset to the default environment. If you omit target, the current environment is printed on standard error.

You may specify either -sh or -csh on a single sde-target invocation, but not both. If you specify neither, sde-target reads the environment variable SHELL (defined under login(1)) to determine which shell to use. If this method fails, an error is reported.

Target names a directory in /usr/sde.

EXAMPLES

sde-target Print the current SDE target.

sde-target -csh default
Output csh(1) commands to reset the SDE target to the default environment.

sde-target -sh m88kbcs
Output sh(1) commands to set the SDE target to the environment named m88kbcs.

FILES

/usr/sde/$TARGET_BINARY_INTERFACE
Root of the target SDE domain.

~/.cshrc User's C shell alias for sde-target.

`$HOME/.profile` User's Bourne shell function for `sde-target`.

DIAGNOSTICS

unknown shell The shell was not specified and could not be determined.

no such target The given target does not exist.

SEE ALSO

csh(1), **sh**(1), **ld**(1), **sdetab**(4), **sde**(5).

NOTE

It is not possible to establish an environment from `make`(1) directly. `Sde-target` must be used before invoking `make`. It is possible when using super-makes to do this automatically.

NAME

`sifilter` – preprocess MC88100 assembly language

SYNOPSIS

`sifilter` [*options*] [*input*] [*output*]

DESCRIPTION

`Sifilter` manipulates MC88100 assembly language source code from *input* to work around known problems in the MC88100 silicon. `Sifilter` is normally invoked transparently by the assembler `/bin/as` but can be used directly for testing purposes. The program can be expected to disappear when silicon is sufficiently mature.

Input and *output* are normally omitted, defaulting to standard input and output paths. Filenames may be specified for either path, and a dash (–), denoting standard input, may be used as a place holder for *input*.

The translations performed by `sifilter` are controlled by the *switches* listed below. The assembler `/bin/as` sets the "standard" option, `-r`. Since each revision of the silicon requires a different set of workarounds, the actual behavior of the "standard" option may vary.

Switches

`a` Insert a `trap-not-taken` (tb1 0,r0,511) after each `ld` or `ld.d`.

`b` Split each `st.d` into an equivalent sequence of two `st` instructions.

`c` Do not pass comment lines through to the output.

`d` Issue each `st` or `st.d` twice.

`D` Synthesize immediate operands of `div` instructions.

`e` Enable literal synthesis. See **Literal Synthesis** below.

`F` Warn about use of double precision source operands in floating point instructions.

`h` Produce code that converts double-precision floating-point operands to single-precision operands before performing floating-point operations. The conversion checks for values outside the range representable in single precision and simulates an illegal instruction trap when conversion is not possible.

`l` Split each `ld.d` into an equivalent sequence of two `ld` instructions.

`l` Split each `ld.d` into an equivalent sequence of two `ld` instructions.

`p` Insert a *trap-not-taken* before each `st` or `st.d`.

`q` Insert a dummy `ld` before each `ld`. A dummy load is a load in which the destination register is r0. The source operands in a dummy load are the same as those in the actual load which follows.

`r` Perform a "standard" set of fixes for current silicon. Check the DG/UX release notice to determine the behavior of the current `sifilter` on the system.

`s` Produce a statistics dump on the standard error path on termination.

`t` Synthesize immediate operands of `div` and `mul` instructions which have any of the high 5 bits set.

`v` Displays a version identification message and exits immediately.

v A single v enables "verbose" mode, in which various messages detailing actions taken by `sifilter` are output as comment lines. Two or more instances of v in the option string generates a comment line containing the current location counter value before each source line.

y Insert a `no-op` after each `trap-not-taken` generated by the z option. If z has not been specified, this option has no effect.

z Insert a `trap-not-taken` after each `st` or `st.d`.

Defaults

All switches default to "off". `Sifilter` performs the following transformations regardless of the option switches specified.

- addu and subu instructions with operands r31,r31,lit32 where "lit32" is a constant whose value is greater than 64K are replaced with an equivalent sequence.
- Floating point instructions involving double operands may be moved if they would otherwise fall at the end of a cache line.

Literal Synthesis

Since `sifilter` must maintain an accurate location counter, it must perform the same fixups for "lit16" operands that would normally be done by a linker performing literal synthesis.

Instructions with lit16 operands whose value cannot be determined by `sifilter` (for example, a label), or whose value would require more than 16 bits, are replaced with an equivalent sequence. This is called "literal synthesis", since a 32-bit value is "synthesized" in a register from the literal.

There are two forms of literal synthesis. The short form:

```
or.u  r29,r0,hi16(lit16)
op    rd,r29,lo16(lit16)
```

is used for the `add`, `addu`, `ld`, `lda`, `or`, `st`, `xmem`, and `xor` instructions (in all their variations) when the source register is r0. When the source register is other than r0, these instructions are expanded into the long form:

```
or.u    r29,r0,hi16(lit16)
or      r29,r29,lo16(lit16)
op      rd,rs,r29
```

Instructions which always are expanded with the long form are all the variations on `and`, `cmp`, `div`, `divu`, `mask`, `mul`, `sub`, `subu`, and `tbnd`.

No literal synthesis is done unless the e option has been specified.

Scratch Registers

Some of the fixups performed by `sifilter` require one or two scratch registers (split `ld.d` or `st.d`, `addu`, `subu`, and floats).

Scratch registers are taken from the set r26–r29.

SEE ALSO

as(1), cc(1).

NOTE

Use of `sifilter` should be coordinated with the revision of silicon on the target machine and the revision of the DG/UX kernel. See the DG/UX release notice for details.

NAME

`size` – print section sizes of object files

SYNOPSIS

`size [-n] [-f] [-o] [-x] [-V] [-F] [-a]` *file* ...

DESCRIPTION

The `size` command produces section or segment size information in bytes (decimal) for each loaded section in the specified object files. `Size` prints out the size of the text, data, and bss (uninitialized data) segments (or sections) and their total. For an archive file, `size` displays this information for each member of the archive.

When calculating segment information, `size` prints out the total file size of the non-writable segments, the total file size of the writable segments, and the total memory size of the writable segments minus the total file size of the writable segments.

If it cannot calculate segment information, `size` calculates section information. When calculating section information, it prints out the total size of sections that are allocatable, non-writable, and not NOBITS, the total size of the sections that are allocatable, writable, and not NOBITS, and the total size of the writable sections of type NOBITS. (NOBITS sections do not actually take up space in the *file*.)

If `size` cannot calculate either segment or section information, it prints an error message and stops processing the file.

Options are:

`-n` Include sections not loaded when calculating the sizes.

`-f` Produce full output; print the size of every loaded section, followed by the section name in parentheses.

`-o` Print numbers in octal, not decimal.

`-x` Print numbers in hexadecimal, not decimal.

`-V` Print, on standard error, the version information about the `size` command being executed.

`-F` Print the size of each loadable segment, the permission flags of the segment, then the total of the loadable segment sizes; this option is accepted only for ELF objects. If there is no segment data, `size` prints an error message and stops processing the file.

`-a` Print a variety of information about components of a common object (COFF) file, including sizes of the file header, optional header, section headers, debug symbols, compiler-generated symbols, local symbols, global symbols, string table, and padding.

EXAMPLES

The examples below are typical `size` output.

`size` *file*	`2724 + 88 + 0 = 2812`
`size -f` *file*	`26(.text) + 5(.init) + 5(.fini) = 36`
`size -F` *file*	`2724(r-x) + 88(rwx) + 0(rwx) = 2812`

DIAGNOSTICS

`size:` *name*`: cannot open`
if *name* cannot be read.

`size:` *name*`: bad magic`
if *name* is not an appropriate object file.

SEE ALSO

as(1), cc(1), ld(1), a.out(4), ar(4).

CAVEAT

Since the size of bss sections is not known until link-edit time, size will not give the true total size of pre-linked objects.

NAME

sno – SNOBOL interpreter and compiler

SYNOPSIS

sno [*files*]

DESCRIPTION

Sno is a SNOBOL compiler and interpreter (with slight differences). sno obtains input from the concatenation of the named *files* and the standard input. All input through a statement containing the label end is considered program and is compiled. The rest is available to syspit.

Sno differs from SNOBOL in the following ways:

There are no unanchored searches. To get the same effect:

a ** b	unanchored search for *b*.
a *x* b = x c	unanchored assignment

There is no back referencing.

x = "abc"	
a *x* x	is an unanchored search for abc.

Function declaration is done at compile time by the use of the (non-unique) label define. Execution of a function call begins at the statement following the define. Functions cannot be defined at run time, and the use of the name define is preempted. There is no provision for automatic variables other than parameters. Examples:

```
define f( )
define f(a, b, c)
```

All labels except define (even end) must have a non-empty statement.

Labels, functions and variables must all have distinct names. In particular, the non-empty statement on end cannot merely name a label.

If start is a label in the program, program execution will start there. If not, execution begins with the first executable statement; define is not an executable statement.

There are no built-in functions.

Parentheses for arithmetic are not needed. Normal precedence applies. Because of this, the arithmetic operators / and * must be set off by spaces.

The right side of assignments must not be empty.

Either ' or " may be used for literal quotes.

The pseudo-variable sysppt is not available.

SEE ALSO

awk(1).

NAME

strip – strip non-executable information from an object file

SYNOPSIS

strip [-l] [-x] [-b] [-r] [-c] [-V] *filename* ...

DESCRIPTION

The strip command strips the symbol table, string table, and line number information from object files, including archives. [See a.out(4)].

After stripping, no symbolic debugging is possible for that file, although a core file produced by a stripped executable can be symbolically debugged if an unstripped copy of the executable is also available. [See sdb(1)]. Normally this command is run only on production modules that have already been debugged and tested.

If strip is executed on a common archive file (see ar(4)) the archive symbol table will be removed. The archive symbol table must be restored by executing ar(1) with the -ts option before the archive can be link-edited by ld(1). Strip generates appropriate warning messages when this situation arises.

Strip takes these options:

-V Print, on the standard error output, the version of strip being executed.

-l Strip only line number information.

-x Do not strip the symbol table from an ELF object file; do not strip static or external symbol information from a COFF object file.

These options are meaningful only when stripping a COFF object file (they are ignored when stripping an ELF object file):

-b Same as the -x option, but also do not strip scoping information (e.g., beginning and end of block delimiters).

-r Do not strip static or external symbol information or relocation information.

-c Strip only compiler-generated symbols.

If there are any relocation entries in a COFF object file and any symbol table information is to be stripped, except by -c, strip complains and terminates without stripping *filename* unless the -r flag is used. If -c is used and there are relocation entries in the COFF object file for compiler generated symbols, strip complains and terminates without stripping.

This command reduces the file storage overhead taken by the object file.

FILES

TMPDIR/strp*	temporary files
TMPDIR	usually /usr/tmp, but can be redefined by setting the environment variable TMPDIR [see tempnam() in tmpnam(3S)].

DIAGNOSTICS

strip: *name*: cannot be read
: if *name* cannot be opened or is too short to be an object file.

strip: *name*: bad magic
: if *name* is not an appropriate object file.

strip: name: relocation entries present; cannot strip
: if *name* contains relocation entries and the -r flag is not used, the symbol table information cannot be stripped.

SEE ALSO

`ar`(1), `as`(1), `cc`(1), `size`(1), `a.out`(4), `ar`(4).

NOTES

The symbol table section will not be removed if it is contained within a segment, or the file is either a relocatable or dynamic shared object.

The line number and debugging sections will not be removed if they are contained within a segment, or their associated relocation section is contained within a segment.

NAME

`tsort` – topological sort

SYNOPSIS

`tsort` [*file*]

DESCRIPTION

`Tsort` produces on the standard output a totally ordered list of items consistent with a partial ordering of items mentioned in the input *file*. If no *file* is specified, the standard input is used.

The input consists of pairs of items (nonempty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

DIAGNOSTICS

Odd data: the input file has an odd number of fields.

SEE ALSO

`lorder`(1).

NAME

unget – undo a previous get of an SCCS file

SYNOPSIS

unget [–r*SID*] [–s] [–n] *files*

DESCRIPTION

Unget undoes the effect of a get –e done before creating the intended new delta. If a directory is named, unget treats each file in the directory as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of – is given, the standard input is read with each line taken as the name of an SCCS file to be processed.

Options apply independently to each named file.

–r*SID* Uniquely identifies which delta is no longer intended. get would specify this as the new delta. The use of this option is necessary only if two or more outstanding gets for editing on the same SCCS file were done by the same person (login name). A diagnostic results if the specified *SID* is ambiguous, or if it is necessary and omitted on the command line.

–s Suppresses the printout, on the standard output, of the intended delta's *SID*.

–n Retains the retrieved file, which would normally be removed from the current directory.

DIAGNOSTICS

Use help(1) for explanations.

SEE ALSO

delta(1), get(1), help(1), sact(1).

NAME

val – validate SCCS file

SYNOPSIS

val -

val [-s] [-r*SID*] [-m*name*] [-y*type*] *files*

DESCRIPTION

val determines if the specified *file* is an SCCS file meeting the characteristics specified by the optional argument list. Arguments to val may appear in any order. The arguments consist of options and named files.

val has a special argument, –, that reads the standard input until an end-of-file condition is detected. Each line read is independently processed as if it were a command line argument list.

val generates diagnostic messages on the standard output for each command line and file processed, and also returns a single 8–bit code upon exit as described below.

The options are listed below. The effects of any option apply independently to each named file on the command line.

–s
: Silences the diagnostic message normally generated on the standard output for any error that is detected while processing each named file on a given command line.

–r*SID*
: The argument value *SID* (*S*CCS *ID*entification String) is an SCCS delta number. A check is made to determine if the *SID* is ambiguous (e. g., -r1 is ambiguous because it physically does not exist but implies 1.1, 1.2, etc., which may exist) or invalid (e. g., -r1.0 or -r1.1.0 are invalid because neither case can exist as a valid delta number). If the *SID* is valid and not ambiguous, a check is made to determine if it actually exists.

–m*name*
: The argument value *name* is compared with the SCCS %M% keyword in *file*.

–y*type*
: The argument value *type* is compared with the SCCS %Y% keyword in *file*.

The 8–bit code returned by val is a disjunction of the possible errors, i. e., can be interpreted as a bit string where (moving from left to right) set bits are interpreted as follows:

bit 0 = missing file argument
bit 1 = unknown or duplicate option
bit 2 = corrupted SCCS file
bit 3 = cannot open file or file not SCCS
bit 4 = *SID* is invalid or ambiguous
bit 5 = *SID* does not exist
bit 6 = %Y%, –y mismatch
bit 7 = %M%, –m mismatch

Note that val can process two or more files on a given command line and in turn can process multiple command lines (when reading the standard input). In these cases an aggregate code – a logical OR of the codes generated for each command line and file processed – is returned.

DIAGNOSTICS

Use help(1) for explanations.

SEE ALSO

admin(1), delta(1), get(1), help(1), prs(1).

BUGS

val can process up to 50 files on a single command line. Any number above 50 will trigger an error.

NAME

valtools – introduction to validation tools

DESCRIPTION

The valtool commands are generally used in shell programming. These commands will prompt for and validate user input. They generally define, among other things, a prompt message, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return). All valtool commands begin with a *ck* prefix.

Visual tool modules are generally linked to the valtool commands. They have *err* (which formats and displays an error message), *help* (which formats and displays a help message), and *val* (which validates a response) prefixes. For example, the `ckpath`(1) command has the following links: **errpath**, **helppath**, and `valpath`, which are used to display an error message, help message, and validate a path.

The following is a list of the available valtool commands with a short description:

`ckdate` prompt for, validate and return a date in the specified format.

`ckgid` prompt for, validate and return an existing group name.

`ckint` prompt for, validate and return an integer value.

ckitem build a menu; prompt for, validate and return a menu item.

`ckkeywd` prompt for, validate and return a keyword from a list of specified keywords.

`ckpath` prompt for, validate and return a pathname that meets the specified criteria.

`ckrange` prompt for, validate and return an integer value between lower and upper bounds.

`ckstr` prompt for, validate and return a string that matches a regular expression.

`cktime` prompt for, validate and return a time value in the specified format.

`ckuid` prompt for, validate and return an existing user login name.

`ckyorn` prompt for, validate and return a yes or no value.

EXIT CODES

All valtool commands exit with code 0 upon successful execution.

SEE ALSO

sh(1), **ckdate**(1), **ckgid**(1), **ckint**(1), **ckitem**(1), **ckkeywd**(1), **ckpath**(1), **ckrange**(1), **ckstr**(1), **cktime**(1), **ckuid**(1), **ckyorn**(1).

NAME

vc – version control

SYNOPSIS

vc [–a] [–t] [–c*char]* [–s] [*keyword*=*value* ... *keyword*=*value*]

DESCRIPTION

The vc command copies lines from the standard input to the standard output under control of its *arguments* and *control statements* encountered in the standard input. User-declared *keywords* may be replaced by their string *value* when they appear in plain text and/or control statements.

The copying of lines from the standard input to the standard output is conditional, based on tests (in control statements) of keyword values specified in control statements or as vc command arguments.

A control statement is a single line beginning with a control character, except as modified by the –t keyletter (see below). The default control character is colon (:), except as modified by the –c keyletter (see below). Input lines beginning with a backslash (\) followed by a control character are not control lines and are copied to the standard output with the backslash removed. Lines beginning with a backslash followed by a non-control character are copied in their entirety.

A keyword is composed of 9 or fewer alphanumerics; the first must be alphabetic. A value is any ASCII string that can be created with ed(1); a numeric value is an unsigned string of digits. Keyword values may not contain blanks or tabs.

Replacement of keywords by values is done whenever a keyword surrounded by control characters is encountered on a version control statement. The –a keyletter (see below) forces replacement of keywords in *all* lines of text. You can include an uninterpreted control character in a value by preceding it with \. If a literal \ is desired, then it too must be preceded by \.

Keyletter Arguments

–a
: Forces replacement of keywords surrounded by control characters with their assigned value in *all* text lines, not just in vc statements.

–t
: All characters from the beginning of a line up to and including the first *tab* character are ignored for the purpose of detecting a control statement. If one is found, all characters up to and including the tab are discarded.

–c*char*
: Specifies a control character to be used in place of :.

–s
: Silences warning messages (not error) that are normally printed on the diagnostic output.

Version Control Statements

:dcl keyword[, ..., keyword]

Declares keywords. All keywords must be declared.

:asg keyword=value

Assigns values to keywords. An asg statement overrides the assignment for the corresponding keyword on the vc command line and all previous asg's for that keyword. Keywords declared, but not assigned values, have null values.

:if condition

⋮

:end

Skips lines of the standard input. If the condition is true all lines between the *if* statement and the matching *end* statement are copied to the standard output. If the condition is false, all intervening lines are discarded, including control statements. Note that intervening *if* statements and matching *end* statements are recognized only for maintaining the proper *if-end* matching.

The syntax of a condition is:

cond	::= ["not"] *or*
or	::= *and* | *and* "|" *or*
and	::= *exp* | *exp* "&" *and*
exp	::= "(" *or* ")" | *value op value*
op	::= "=" | "!=" | "<" | ">"
value	::= <arbitrary ASCII string> | <numeric string>

The available operators and their meanings are:

=	Equal
!=	Not equal
&	And
|	Or
>	Greater than
<	Less than
( )	Used for logical groupings
not	May occur only immediately after the *if*, and when present, inverts the value of the entire condition

The > and < operate only on unsigned integer values (e.g., : 012 > 12 is false). All other operators take strings as arguments (e.g., : 012 != 12 is true). The precedence of the operators (from highest to lowest) is:

= != > < All of equal precedence

&

|

Use parentheses to alter the order of precedence.

Separate values from operators or parentheses by at least one blank or tab.

: :text

Replaces keywords on lines that are copied to the standard output. The two leading control characters are removed, and keywords surrounded by control characters in text are replaced by their value before the line is copied to the output file. This action is independent of the **−a** keyletter.

:on

:off

Turns on or off keyword replacement on all lines.

:ctl char

Changes the control character to *char*.

:msg message

Prints the given message on the diagnostic output.

:err message

Prints the given message followed by:

ERROR: err statement on line ... (915)

on the diagnostic output. vc halts execution, and returns an exit code of 1.

DIAGNOSTICS

Use help(1) for explanations.

EXIT CODES

0–Normal

1–Any error

SEE ALSO

ed(1), help(1).

NAME

`what` – identify SCCS files

SYNOPSIS

`what` [`-s`] *files*

DESCRIPTION

`What` searches the given files for all occurrences of the pattern that `get`(1) substitutes for %Z% (this is `@(#)` at this printing) and prints out what follows until the first `"`, `>`, new-line, `\`, or null character. For example, if the C program in file `f.c` contains

```
char ident[] = "@(#)identification information";
```

and `f.c` is compiled to yield `f.o` and `a.out`, then the command

```
what f.c f.o a.out
```

will print identification information for `f.c`, `f.o`, and `a.out`.

`What` is for use with the SCCS command `get`(1), which automatically inserts identifying information; but you can also use it where the information is inserted manually. Only one option exists:

`-s` Quit after finding the first occurrence of the pattern in each file.

DIAGNOSTICS

Exit status is 0 if any matches are found, otherwise it's 1. Use `help`(1) for explanations.

SEE ALSO

`get`(1), `help`(1).

BUGS

An unintended occurrence of the pattern `@(#)` could be found by chance, but this usually causes no harm.

NAME

xstr – extract strings from C programs to implement shared strings

SYNOPSIS

xstr [-c] [-] [*file*]

DESCRIPTION

Xstr maintains a file strings into which strings in component parts of a large program are hashed. These strings are replaced with references to this common area. This serves to implement shared constant strings, which are most useful if they are also read-only.

The command

    xstr -c name

will extract the strings from the C source in *name*, replacing string references by expressions of the form (&xstr[number]) for some number. An appropriate declaration of xstr is prepended to the file. The resulting C text is placed in the file x.c, to then be compiled. The strings from this file are placed in the strings data base if they are not there already. Repeated strings and strings which are suffixes of existing strings do not cause changes to the data base.

After all components of a large program have been compiled, a file xs.c declaring the common xstr space can be created by a command of the form

    xstr

This xs.c file should then be compiled and loaded with the rest of the program. If possible, the array can be made read-only (shared), saving space and swap overhead.

Xstr can also be used on a single file. A command

    xstr name

creates files x.c and xs.c as before, without using or affecting any strings file in the same directory.

It may be useful to run xstr after the C preprocessor if any macro definitions yield strings or if there is conditional code which contains strings which may not, in fact, be needed. Xstr reads from its standard input when the argument '–' is given. An appropriate command sequence for running xstr after the C preprocessor is:

```
cc -E name.c | xstr -c -
cc -c x.c
mv x.o name.o
```

Xstr does not touch the file strings unless new items are added, thus make can avoid remaking xs.o unless truly necessary.

FILES

strings	Data base of strings
x.c	Massaged C source
xs.c	C source for definition of array 'xstr'
/tmp/xs*	Temp file when 'xstr name' doesn't touch strings

SEE ALSO

mkstr(1).

NOTE

If a string is a suffix of another string in the data base, but the shorter string is seen first by xstr both strings will be placed in the data base, when just placing the longer one there will do.

NAME

yacc – yet another compiler-compiler

SYNOPSIS

yacc [-vVdlt] [-Q[y|n]] *file*

DESCRIPTION

The yacc command converts a context-free grammar into a set of tables for a simple automaton that executes an LALR(1) parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output file, y.tab.c, must be compiled by the C compiler to produce a program yyparse. This program must be loaded with the lexical analyzer program, yylex, as well as main and yyerror, an error handling routine. These routines must be supplied by the user; the lex(1) command is useful for creating lexical analyzers usable by yacc.

-v Prepares the file y.output, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

-d Generates the file y.tab.h with the #define statements that associate the yacc-assigned "token codes" with the user-declared "token names." This association allows source files other than y.tab.c to access the token codes.

-l Specifies that the code produced in y.tab.c will not contain any #line constructs. This option should only be used after the grammar and the associated actions are fully debugged.

-Q[y|n] The -Qy option puts the version stamping information in y.tab.c. This allows you to know what version of yacc built the file. The -Qn option (the default) writes no version information.

-t Compiles runtime debugging code by default. Runtime debugging code is always generated in y.tab.c under conditional compilation control. By default, this code is not included when y.tab.c is compiled. Whether or not the -t option is used, the runtime debugging code is under the control of YYDEBUG, a preprocessor symbol. If YYDEBUG has a non-zero value, then the debugging code is included. If its value is zero, then the code will not be included. The size and execution time of a program produced without the runtime debugging code will be smaller and slightly faster.

-V Prints on the standard error output the version information for yacc.

FILES

y.output	
y.tab.c	
y.tab.h	defines for token names
yacc.tmp,	
yacc.debug, yacc.acts	temporary files
LIBDIR/yaccpar	parser prototype for C programs
LIBDIR	usually /usr/ccs/lib

SEE ALSO

lex(1).
The "yacc" chapter in the *Programmer's Guide: ANSI C and Programming Support Tools*.

DIAGNOSTICS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard error output; a more detailed report is found in the `y.output` file. Similarly, if some rules are not reachable from the start symbol, this instance is also reported.

NOTES

Because file names are fixed, at most one `yacc` process can be active in a given directory at a given time.

End of Chapter

Chapter 2
System Calls

This chapter contains in printed form the online manual entries for DG/UX system calls. The first entry, **intro**(2), gives an introduction to DG/UX system calls. The rest of the entries are in alphabetical order.

NAME

`intro` – introduction to system calls and error numbers

SYNOPSIS

`#include <errno.h>`

DESCRIPTION

This chapter describes all of the system calls. This introduction is divided into two parts: DEFINITIONS and DIAGNOSTICS.

The DEFINITIONS section identifies important system abstractions and describes them briefly in terms of their representation in the system (that is, the *superuser* abstraction is described in terms of its identity within the system: a superuser process is one with an effective user id of 0; it has special privileges). A summary of definitions appears at the head of the section; both the summary and the individual entries are grouped into categories. The categories are: processes, files, messages, semaphores, shared memory, interprocess communications primitives, UNIX communications domain, and Internet communications domain. Most entries are short and do not suggest the programming contexts in which you use the system calls mentioned; they generally refer you to one or more individual system call descriptions in the manual. However, the **Interprocess Communications Primitives** section is a rather extensive discussion. It is taken from the *4.2BSD System Manual* by Joy, Cooper, Fabry, Leffler, McKusick, and Mosher, University of California, Berkeley, Berkeley CA.

The DIAGNOSTICS section lists the entire set of error conditions by number, name, and description. At the end of the DIAGNOSTICS section is a discussion of implementation-dependent constants that are referenced in the discussions of individual calls.

DEFINITIONS

Processes

Process ID

A positive integer used to identify a process; each process in the system has a unique process ID. The range of this ID is from 0 to PID_MAX (30,000).

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).

Process Group Leader

A process group leader is a process that creates a new process group. The process group ID of a process group is equal to the process group ID of the process group leader.

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. The group ID can be used to terminate a group of related processes when one of the processes in the group is terminated; see `exit`(2) and `signal`(2).

Real User ID and Real Group ID

Each user allowed on the system is identified by a positive integer called a user ID.

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

An active process has a real user ID and real group ID that are set to the user ID and group ID, respectively, of the user who created the process.

Effective User ID and Effective Group ID

Each 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).

Superuser

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

Special Processes

Processes with a process ID of 0 or 1 are special processes and are referred to as `proc0` and `proc1`.

`Proc0` is the scheduler. `Proc1` is the initialization process (init). `Proc1` is the ancestor of every other process in the system; it controls the process structure.

Files

Descriptor

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

A process may have no more than OPEN_MAX descriptors (0 to (OPEN_MAX-1)) open simultaneously, unless the RLIMIT_NOFILE command of `setrlimit`(2) has been used to increase the limit.

The descriptor is used as an argument by calls such as `read`(2), `write`(2), `ioctl`(2), `send`(2), `recv`(2), and `close`(2).

The descriptor is known as the *file descriptor* in System V.

Filename

A filename is a character string that names an ordinary file, special file or directory. Filenames can be up to 255 characters.

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

Avoid using `*`, `?`, `@`, `#`, `$`, `^`, `&`, `(`, `)`, `` ` ``, `|`, `;`, `"`, `<`, `>`, `[`, `\`, `]`, `!`, `~` `{` or `}` as part of filenames, since the shells attach special meaning to them. Avoid using `-` as the first character of a filename, since `-` is used to begin an option in a command line. Also avoid using unprintable characters in filenames. See `sh`(1) and `csh`(1).

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 filename.

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; see `chroot`(2).

File Access Permissions

Every file in the file system has a set of access permissions, which determine whether a process may perform a requested operation on the file. For example, opening a file for writing is an operation subject to file access permissions.

Every file has three classes of access permissions: owner (user), group, and other. The classes identify types of users: the *owner* of a file, a defined *group* of users, and all *other* users.

Each class has its own set of three types of access permissions: *read* (r), *write* (w), and *execute* (x). A file's access permissions are set when the file is created. They can be masked upon creation if `umask`(2) is in effect, and they can be changed explicitly with `chmod`(2), `chown`(2), or `chgrp`(2). When an access check is made, the system decides if permission should be granted by comparing the file's access permissions and the calling process's access information.

All three permissions (read, write, and execute/search) on a file are granted to a calling process if one or more of the following are true:

> The effective user ID of the calling process is superuser.
>
> The effective user ID of the calling process matches the user ID of the owner of the file and the appropriate access bits of the `owner` portion (0700) of the file mode is set.
>
> The effective user ID of the calling process does not match the user ID of the owner of the file, the effective group ID of the calling process matches the group of the file, and the appropriate access bits of the `group` portion (070) of the file mode is set.
>
> The effective user ID of the calling process does not match the user ID of the owner of the file, the effective group ID of the calling process does not match the group ID of the file, and the appropriate access bits of the `other` portion (07) of the file mode is set.

Otherwise, permissions are denied on the basis of permission values in the file mode.

Messages

Message Queue Identifier

A message queue identifier (msqid) is a unique positive integer created by a `msgget`(2) system call. The maximum number of *msqid*s allowed is configurable. The default is 50. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

```
struct  ipc_perm msg_perm;   /* operation permission struct */
ushort  msg_qnum;            /* number of msgs on q */
ushort  msg_qbytes;          /* max number of bytes on q */
pid_t   msg_lspid;           /* pid of last msgsnd operation */
pid_t   msg_lrpid;           /* pid of last msgrcv operation */
time_t  msg_stime;           /* last msgsnd time */
time_t  msg_rtime;           /* last msgrcv time */
time_t  msg_ctime;           /* last change time */
                             /* all times are in secs since */
                             /* 00:00:00 GMT, Jan. 1, 1970 */
```

`Msg_perm` is an `ipc_perm` structure, as declared in `ipc.h`, that specifies the message operation permission (see below). This structure includes the following members:

```
uid_t          cuid;  /* creator user id */
gid_t          cgid;  /* creator group id */
uid_t          uid;   /* user id */
gid_t          gid;   /* group id */
unsigned long  mode;  /* r/w permission */
```

`Msg_qnum` is the number of messages currently on the queue. `Msg_qbytes` is the maximum number of bytes allowed on the queue. `Msg_lspid` is the process ID of the last process that performed a `msgsnd` operation. `Msg_lrpid` is the process ID of the last process that performed a `msgrcv` operation. `Msg_stime` is the time of the last `msgsnd` operation, `msg_rtime` is the time of the last `msgrcv` operation, and `msg_ctime` is the time the *msgid* was created by `msgget`(2) or of the last `msgctl`(2) operation that changed a member of the above structure.

Message Operation Permissions

In the `msgop`(2) and `msgctl`(2) system call descriptions, the permission required for an operation is given as *token*. *Token* is the type of permission needed interpreted as follows:

00400	Read by user
00200	Write by user
00060	Read, write by group
00006	Read, write by others

Read and write permissions on a *msqid* are granted to a process if one or more of the following are true:

The effective user ID of the process is superuser.

The effective user ID of the process matches `msg_perm.[c]uid` in the data structure associated with *msqid* and the appropriate bit of the `user` portion (0600) of `msg_perm.mode` is set.

The effective user ID of the process does not match msg_perm.[c]uid, the effective group ID of the process matches msg_perm.[c]gid, and the appropriate bit of the group portion (060) of msg_perm.mode is set.

The effective user ID of the process does not match msg_perm.[c]uid, the effective group ID of the process does not match msg_perm.[c]gid, and the appropriate bit of the other portion (06) of msg_perm.mode is set.

Otherwise, the corresponding permissions are denied.

Semaphores

Semaphore Identifier

A semaphore identifier (semid) is a unique positive integer created by a semget(2) system call. The maximum numbers of identifiers is configurable; the default is 10. Each has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and contains the following members:

```
struct   ipc_perm sem_perm;   /* operation permission struct */
ushort   sem_nsems;           /* number of sems in set */
time_t   sem_otime;           /* last operation time */
time_t   sem_ctime;           /* last change time */
                              /* all times are in secs since */
                              /* 00:00:00 GMT, Jan. 1, 1970 */
```

Sem_perm is an ipc_perm structure (as defined in ipc.h) that specifies the semaphore operation permission (see below). This structure includes the following members:

```
uid_t           cuid;  /* creator user id */
gid_t           cgid;  /* creator group id */
uid_t           uid;   /* user id */
gid_t           gid;   /* group id */
unsigned long   mode;  /* r/a permission */
```

The value of sem_nsems is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a *sem_num*. Sem_num values run sequentially from 0 to the value of sem_nsems minus 1. Sem_otime is the time of the last semop(2) operation, and sem_ctime is the time the *sem_id* was created by semget(2) or of the last semctl(2) operation that changed a member of the above structure.

A semaphore is a data structure that contains the following members:

```
ushort   semval;   /* semaphore value */
pid_t    sempid;   /* pid of last operation  */
ushort   semncnt;  /* # awaiting semval > current value */
ushort   semzcnt;  /* # awaiting semval = 0 */
```

Semval is a non-negative integer in the range 0 to PID_MAX (30,000). Sempid is equal to the process ID of the last process that performed a semaphore operation on this semaphore. Semncnt is the number of processes waiting for this semaphore's semval to exceed its current value. Semzcnt is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become zero.

Semaphore Operation Permissions

In the semop(2) and semctl(2) system call descriptions, the permission required for an operation is given as *token*. *Token* is interpreted as follows:

00400	Read by user
00200	Alter by user
00060	Read, alter by group
00006	Read, alter by others

Read and alter permissions on a semid are granted to a process if one or more of the following are true:

The effective user ID of the process is superuser.

The effective user ID of the process matches `sem_perm.[c]uid` in the data structure associated with *semid*, and the appropriate bit of the `user` portion (0600) of `sem_perm.mode` is set.

The effective user ID of the process does not match `sem_perm.[c]uid`, the effective group ID of the process matches `sem_perm.[c]gid`, and the appropriate bit of the `group` portion (060) of `sem_perm.mode` is set.

The effective user ID of the process does not match `sem_perm.[c]uid`, the effective group ID of the process does not match `sem_perm.[c]gid`, and the appropriate bit of the `other` portion (06) of `sem_perm.mode` is set.

Otherwise, the corresponding permissions are denied.

Shared Memory

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. The data structure is referred to as *shmid_ds* and contains the following members:

```
struct  ipc_perm shm_perm;   /* operation permission struct */
int     shm_segsz;           /* size of segment */
pid_t   shm_cpid;            /* creator pid */
pid_t   shm_lpid;            /* pid of last operation */
ushort  shm_nattch;          /* number of current attaches */
time_t  shm_atime;           /* last attach time */
time_t  shm_dtime;           /* last detach time */
time_t  shm_ctime;           /* last change time */
                             /* all times are in secs since */
                             /* 00:00:00 GMT, Jan. 1, 1970 */
```

`Shm_perm` is an `ipc_perm` structure that specifies the shared memory operation permission (see below). This structure includes the following members:

```
uid_t          cuid;  /* creator user id */
gid_t          cgid;  /* creator group id */
uid_t          uid;   /* user id */
gid_t          gid;   /* group id */
unsigned long  mode;  /* r/w permission */
```

`Shm_segsz` specifies the size in bytes of the shared memory segment. `Shm_cpid` is the process ID of the process that created the shared memory identifier. `Shm_lpid` is the process ID of the last process that performed a `shmat(2)` or `shmdt(2)` operation. `Shm_nattch` is the number of processes that currently have this segment attached. `Shm_atime` is the time of the last `shmat` operation, `shm_dtime` is the time of the last `shmdt` operation, and `shm_ctime` is the time the *shm_id* was created by `shmget(2)` or of the last `shmctl(2)` operation that changed one of the members of the above structure.

Shared Memory Operation Permissions

In the descriptions for the shmsys(2) family of system calls, the permission required for an operation is given as *token*, where *token* is interpreted as follows:

00400	Read by user
00200	Write by user
00060	Read, write by group
00006	Read, write by others

Read and write permissions on a *shmid* are granted to a process if one or more of the following are true:

The effective user ID of the process is superuser.

The effective user ID of the process matches `shm_perm.[c]uid` in the data structure associated with *shmid*, and the appropriate bit of the `user` portion (0600) of `shm_perm.mode` is set.

The effective user ID of the process does not match `shm_perm.[c]uid`, the effective group ID of the process matches `shm_perm.[c]gid`, and the appropriate bit of the group portion (060) of `shm_perm.mode` is set.

The effective user ID of the process does not match `shm_perm.[c]uid`, the effective group ID of the process does not match `shm_perm.[c]gid`, and the appropriate bit of the `other` portion (06) of `shm_perm.mode` is set.

Otherwise, the corresponding permissions are denied.

Interprocess Communication Primitives

This section (up to the DIAGNOSTICS section) describes the DG/UX IPC facilities, which are based on the Berkeley UNIX IPC facilities.

Communication Domains

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

NOTE: The "internet" domain is not provided on the standard DG/UX system. This domain is provided only with the DG/UX TCP/IP product.

Socket Types and Protocols

Within a domain, communication takes place between communication endpoints known as *sockets*. Each socket has queues for sending and receiving data; it may exchange data with other sockets within its domain.

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

Each kernel supports some collection of socket types; consult `socket(2)` for more information about the types available and their properties. The basic set of socket types is defined in `<sys/socket.h>`:

```
/* Standard socket types */
```

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

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

Other socket types can be defined.

NOTE: The DG/UX system does not support the SOCK_RDM and SOCK_SEQPACKET types.

Each socket may have a concrete *protocol* associated with it. This protocol is used within the domain to provide the semantics required by the socket type. For example, within the "internet" domain, the SOCK_DGRAM type may be implemented by the UDP user datagram protocol, and the SOCK_STREAM type may be implemented by the TCP transmission control protocol, while no standard protocols to provide SOCK_RDM or SOCK_SEQPACKET sockets exist.

Each kernel supports some number of sets of communications protocols. Each protocol set supports addresses of a certain format. An Address Family is the set of addresses for a specific group of protocols. Each socket has an address chosen from the address family in which the socket was created.

Socket Creation, Naming and Service Establishment

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

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

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

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

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

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

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

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

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

Domains may support sockets with several names.

Accepting Connections

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

> listen(*s*, *backlog*);
> int *s*, *backlog*;

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

An accept call:

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

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

Making Connections

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

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

It is also possible to create connected pairs of sockets without using the domain's name space to rendezvous; this is done with the socketpair call (only in the "unix" communication domain):

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

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

Sending and Receiving Data

Messages may be sent from a socket by:

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

if the socket is not connected or:

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

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

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

and

> *msglen* = recv(*s*, *buf*, *len*, *flags*);
> result in *msglen*; int *s*; result caddr_t *buf*; int *len*, *flags*;

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

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

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

Scatter/Gather and Exchanging Access Rights

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

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

Here `msg_name` and `msg_namelen` specify the source or destination address if the socket is unconnected; `msg_name` may be given as a null pointer if no names are desired or required. The `msg_iov` and `msg_iovlen` describe the scatter/gather locations.

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

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

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

Using Read and Write with Sockets

The normal DG/UX `read` and `write` calls may be applied to connected sockets and translated by the system into `send` and `receive`. A process may operate on a virtual circuit socket, a terminal or a file with blocking or non-blocking input/output operations without distinguishing the descriptor type.

Shutting Down Halves of Full-duplex Connections

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

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

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

Socket and Protocol Options

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

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

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

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

UNIX Communications Domain

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

Types of Sockets

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

Naming

Socket names are strings and appear in the UNIX file system name space. (The DG/UX implementation of the UNIX domain embeds bound sockets in the UNIX file system name space; this is a side effect of the implementation.)

INTERNET Communications Domain

This section describes briefly how the INTERNET domain is mapped to the model described in this section.

Socket Types and Protocols

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

Socket Naming

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

Access Rights Transmission

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

Raw Access

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

DIAGNOSTICS

Most system calls have one or more error returns. An error condition is generally indicated by an otherwise impossible returned value. This value is almost always −1; the individual descriptions specify the details of each error. When an error occurs, an error number is recorded and made available in the external variable `errno`. `Errno` is not cleared on successful calls, so it should be tested only after an error has been indicated.

Each system call description in this manual lists the possible error numbers that the call could return. The descriptions listed in the individual sections may not be identical to the ones listed here; they try to be specific to the particular call's context. The following is a complete general reference list of all error numbers and their names; they are defined in `<errno.h>`.

Numbering

1 EPERM Not owner
: This error usually indicates an attempt to modify a file in some way forbidden except to its owner or to the super-user. It also indicates attempts by ordinary users to do things allowed only to the super-user.

2 ENOENT No such file or directory
: This error occurs when you try to use a pathname that is too long, refer to a file that doesn't exist, or use a path name that includes an invalid directory name (e.g., the directory doesn't exist).

3 ESRCH No such process
: No process can be found corresponding to that specified by the search criteria.

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 resumes after processing the signal, the interrupted system call will return this error condition.

5 EIO I/O error
: Some physical I/O error has occurred.

6 ENXIO No such device or address
: I/O on a special file refers to a subdevice that does not exist, or that extends beyond the limits of the device. It may also occur when a device is not on-line or no disk pack is loaded on a drive.

7 E2BIG Argument list too long
: An argument list longer than ARG_MAX (10240) bytes is presented to a member of the `exec` 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 format (see `a.out`(4)).

9 EBADF Bad file number
: Occurs under any of three conditions: a file descriptor refers to no open file; a read request is made to a file that is open only for writing; a write request is made to a file that is open only for reading.

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

11 EAGAIN or EWOULDBLOCK Resource temporarily unavailable
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, such as a `brk` or `sbrk`, failed because of insufficient memory or swap space. Or, an operation that would cause a process to block was attempted on a object in non-blocking mode (see `ioctl`(2)).

12 ENOMEM Not enough space
The system could not supply the memory required to complete the system call.

13 EACCES Permission denied
Access was attempted to an object for which the caller lacked the required access privilege(s).

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`.

16 EBUSY Device or resource busy
You tried to mount a device that was already mounted or to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). This error will also occur if you try to enable accounting when it is already enabled, or if device or resource requested 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
You tried to link to a file on another device, or rename a file across devices.

19 ENODEV No such device
You tried to apply an inappropriate system call to a device; e.g., read a write-only device.

20 ENOTDIR Not a directory
You gave a non-directory reference where a directory reference 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 open a directory file for writing.

22 EINVAL Invalid argument
Some invalid argument; e.g., dismounting a non-mounted device, mentioning an undefined signal in `signal` or `kill`, reading or writing a file for which `lseek` 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 no more `opens` can be accepted now.

24 EMFILE Too many open files
No process may have more than OPEN_MAX (by default, 64) file descriptors open at a time.

25 ENOTTY Not a character device
You tried to `ioctl(2)` a file that is not a character-special device.

26 ETXTBSY Open Intent conflict
You attempted to do unbuffered I/O on a buffered I/O channel or the converse of that situation.

27 EFBIG File too large
A file exceeded the maximum file size (1,082,201,088 bytes) or ULIMIT; see `ulimit(2)`.

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

29 ESPIPE Illegal seek
An `lseek` was issued to a pipe or socket.

30 EROFS Read-only file system
You tried to modify a file or directory on a device mounted read-only.

31 EMLINK Too many links
You tried to make more than the maximum number of links (LINK_MAX) 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 ECHRNG Channel number out of range

38 EL2NSYNC Level 2 not synchronized

39 EL3HLT Level 3 halted

40 EL3RST Level 3 reset

41 ELNRNG Link number out of range

42 EUNATCH Protocol driver not attached

43 ENOCSI No CSI structure available

44 EL2HLT Level 2 halted

45 EDEADLK Deadlock in lockf

46 ENOLCK No record locks available

50 EBADE Invalid exchange

51 EBADR Invalid request descriptor

52 EXFULL Exchange full

53 ENOANO No anode

54 EBADRQC Invalid request code

55 EBADSLT Invalid slot

56 EDEADLOCK File locking deadlock error

57 EBFONT Bad font file format

60 ENOSTR A streams-only operation was attempted on a non-stream file

61 ENODATA No data (for no-delay I/O)

62 ETIME Operation timed out

63 ENOSR Streams resources are not available

64 ENONET Machine is not on the network

65 ENOPKG Package not installed

66 EREMOTE Cannot mount a file system onto a remote directory

67 ENOLINK The link has been severed

68 EADV Advertise error

69 ESRMNT SRmount error

70 ECOMM Communication error on send

71 EPROTO Protocol error

74 EMULITHOP Multihop attempted

77 EBADMSG Bad message type

78 ENAMETOOLONG File name too long

80 ENOTUNIQ Given logname not unique

81 EBADFD File descriptor invalid for this operation

82 EREMCHG Remote address changed

83 ELIBACC Cannot access a needed shared library

84 ELIBBAD Accessing a corrupted shared library

85 ELIBSCN The .lib section in an executable is corrupted

86 ELIBMAX Attempting to link in too many shared libraries

87 ELIBEXEC Attempting to execute a shared library

89 ENOSYS Function not implemented

90 ELOOP Too many levels of symbolic links

91 ERESTART Restartable system call

128 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 `ioctl`(2)).

129 EALREADY Operation already in progress

130 ENOTSOCK Socket operation on non-socket
A socket-specific operation (such as `bind`(2)) was attempted on a non-socket file.

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

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

133 EPROTOTYPE Protocol wrong type for socket
You specified a protocol that does not support the semantics of the socket type requested. For example, you cannot use the DARPA Internet UDP protocol with type SOCK_STREAM.

134 ENOPROTOOPT Bad protocol option
You specified a bad option in a `getsockopt`(2) or `setsockopt`(2) call.

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

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

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

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

139 EAFNOSUPPORT Address family not supported by protocol family
You used an address incompatible with the requested protocol. For example, you can't always use PUP Internet addresses with DARPA Internet protocols.

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

141 EADDRNOTAVAIL Cannot assign requested address
This error usually results from an attempt to create a socket with an address not on this machine.

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

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

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

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

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

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

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

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

150 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.

151 ETOOMANYREFS
Too many references; cannot splice.

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

153 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.

156 EHOSTDOWN Host is down

157 EHOSTUNREACH No route to host

158 ENOTEMPTY Directory not empty

159 EPROCLIM (Not used in DG/UX)

160 EUSERS Too many users

161 EDQUOT Disk quota exceeded

162 ESTALE Stale NFS file handle

163 EPOWERFAIL Power failure occurred

SEE ALSO

`close(2)`, `connect(2)`, `ioctl(2)`, `open(2)`, `pipe(2)`, `read(2)`, `shutdown(2)`, `ulimit(2)`, `write(2)`, `intro(3)`, `lockf(3C)`, `perror(3C)`.

NAME

accept – accept a connection on a socket

SYNOPSIS

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

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

where:

s	File descriptor of socket listening for connection requests
addr	Structure to receive the address of newly connected peer
addrlen	On input contains the number of bytes available for the peer address; updated to indicate the number of bytes returned

DESCRIPTION

The argument *s* is the file descriptor of a socket that has been:

- Created with the socket system call.
- Bound to an address with bind(2).
- Made to listen for connections with listen(2).

Accept extracts the first connection on the queue of pending connections, creates a new socket of the same type (e.g. SOCK_STREAM) as *s*, and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue and the socket is not marked as non-blocking, accept blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, accept returns an error as described below. The accepted socket, *ns*, will be in the connected state. The original socket *s* remains open listening for more connections.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication is occurring. See related documentation for a descripton of address formats for each domain. *addrlen* is a value/result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. If *addrlen* is zero, the pointer will be ignored. If the address buffer is too small to hold all of the address, the address will be truncated.

This call is used with connection-based socket types, currently with SOCK_STREAM.

A select system call can be issued on a listening socket for notification of connnection requests.

ACCESS CONTROL

None.

RETURN VALUE

ns	The call was successful. *ns* is a non-negative integer that is a descriptor for the accepted socket.
–1	An error occurred. errno is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	The argument *s* is not an active, valid descriptor.
ENOTSOCK	The descriptor references a file, not a socket.
EMFILE	No more user file descriptors available, the per-process limit has been reached.
ENFILE	No more system file descriptors available, the system limit has been reached.
EINVAL	The socket *s* is not in the listen state.
EOPNOTSUPP	The referenced socket is not of type SOCK_STREAM.
EFAULT	The *addr* parameter is not in a writable part of the user address space.
ECONNABORTED	Listen operation aborted by system.
EAGAIN	The socket is marked non-blocking and no connections are present to be accepted.
EINTR	The call was interrupted by a signal.

SEE ALSO

`bind`(2), `connect`(2), `listen`(2), `select`(2), `socket`(2), `inet`(3N), `inet`(6F), `unix_ipc`(6F).

NAME

access – determine the accessibility of a file

SYNOPSIS

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

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

where:

path Address of a pathname naming a file of type ordinary, directory, FIFO, block special, character special, or symbolic link.

amode Access mode bit pattern

DESCRIPTION

Access checks that the calling process has specified access rights to the file. If *path* refers to a symbolic link, the target of the symbolic link is checked, not the symbolic link. The types of access requested are indicated by *amode*, which can have the following values:

F_OK 0
Check the existence of a file.

X_OK 1
Check for execute access. Applied to a directory, execute permission allows the directory to be used in a pathname.

W_OK 2
Check for write access. Applied to a directory, write permission allows the creation and deletion of links in the directory.

R_OK 4
Check for read access. Applied to a directory, read permission allows the contents of the directory to be listed.

Some combination (logical OR) of X_OK, W_OK, and R_OK.

Write access is categorically denied when the file is of type ordinary, directory, or FIFO and resides on a read-only file system device. In this case, errno is set to EROFS.

In all other situations, a process with a real user id of superuser is granted all access rights. Other processes are granted access only if the file's mode gives them all types of access requested.

When determined by the file's mode, access is checked with respect to only one of the owner, group, and other subsets of the mode. If the process's real user id matches the file's user id, the owner bits of the file mode determine access. If the process's real user id doesn't match, but its real group id matches the file's group id or one of the group ids in its group set matches the file's group id, the group bits of the file mode determine access. In all other cases, the 'other' mode bits determine access.

Note that this call does not guarantee that a file is writable or executable if write or execute access is granted. For instance, a directory's mode may give the caller write access, indicating that files may be created in it, but an open of the directory for write intent will fail.

ACCESS CONTROL

The caller must have permission to resolve *path*. This call differs from others in that the process's real user id is used to determine permission to resolve a pathname, rather than its effective user id.

RETURN VALUE

0	The requested access is permitted.
–1	Access to the file is denied. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EACCES	Permission bits of the file mode do not permit the requested access.
EROFS	Write access is requested for a file of type ordinary, directory, or FIFO that resides on a file system device mounted read-only.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

chmod(2), fstat(2), stat(2), stat(5).

NAME

`acct` – enable or disable process accounting

SYNOPSIS

```
#include <unistd.h>

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

where:

path Address of a pathname

DESCRIPTION

This function provides capabilities that are inherently implementation dependent.

`Acct` enables or disables process accounting. If process accounting is enabled, an accounting record will be written on an accounting file for each process that terminates.

Path points to the path name of the accounting file. The accounting file format and the information it contains are implementation dependent.

Process accounting is disabled if *path* is NULL and enabled if *path* is non-NULL.

If errors occur during the acct operation, the status of process accounting is not changed. If errors occur when an accounting record is written, the record may be lost.

ACCESS CONTROL

The effective-user-id of the calling process must be super-user.

RETURN VALUE

0 The accounting file was successfully changed.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EPERM	Permission to enable or disable process accounting is denied to the calling process.
EACCES	The file named by *path* is not an ordinary file.
EISDIR	The named file is a directory.
EROFS	The named file resides on a read-only file system.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.

EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

acct(1M). signal(2), exit(3C), acct(4).

NAME

`adjtime` – correct the time to allow synchronization of the system clock

SYNOPSIS

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

adjtime(delta, olddelta)
struct timeval *delta;
struct timeval *olddelta;
```

where:

delta The name of a structure containing a number of seconds

olddelta The name of a structure containing a number of seconds

DESCRIPTION

`Adjtime` makes small adjustments to the system time, as returned by `gettimeofday`(2), advancing or retarding it by the amount of time specified by the `struct timeval` pointed to by *delta*. The adjustment is gradual. If **delta* represents a negative adjustment, the clock is slowed down by incrementing it more slowly than normal until the correction is complete. If **delta* represents a positive adjustment, the correction is achieved by using a larger than normal increment.

Specify a positive adjustment by placing a non-negative number of seconds in *delta->tv_sec* and a number of microseconds between 0 and 999999 (inclusive) in *delta->tv_usec*. Specify a negative adjustment by placing a negative number of seconds in *delta->tv_sec* and a number of microseconds between 0 and 999999 (inclusive) in *delta->tv_usec*. Note that the number of microseconds must always be non-negative and always acts to widen an advancement or to shorten a delay. For instance, to indicate a delay of 0.7 seconds, place −1 into *delta->tv_sec* and 300000 into *delta->tv_usec*. To indicate an advancement of 7.22 seconds, place 7 into *delta->tv_sec* and 220000 into *delta->tv_usec*.

A time correction from an earlier call to `adjtime` may not be finished when `adjtime` is called again. In this case, the previous time correction is aborted. Further, if *olddelta* is not NULL, then the struct timeval it points to will contain, upon return, the number of microseconds which were still to be corrected from the earlier call.

Note also that setting the time of day does not cancel any time adjustments in progress.

This call may be used by time servers that synchronize the clocks of computers in a local area network. Such time servers would slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time.

ACCESS CONTROL

The effective user id of the calling process must be superuser.

RETURN VALUE

0 The call succeeded.

−1 An error occurred; an error code is stored in the global variable `errno`.

DIAGNOSTICS

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

EFAULT An argument points outside the process's allocated address space.

EPERM The process's effective user id is not that of the super-user.

SEE ALSO

date(1), gettimeofday(2).

NAME

`alarm` – set a process alarm clock

SYNOPSIS

```
#include <unistd.h>

unsigned int alarm (sec)
unsigned int sec;
```

where:

sec The number of real-time seconds to wait before sending SIGALRM to the caller

DESCRIPTION

`Alarm` sets the caller's per-process real-time alarm clock to send the signal SIGALRM to the calling process after the number of real-time seconds specified by *sec* have elapsed. Alarm requests are not stacked; successive calls reset the calling process's alarm clock. If *sec* is 0, any previous alarm request is canceled. `Alarm` uses the ITIMER_REAL interval timer as is used by the `setitimer` system call. The `fork` system call resets the alarm clock in the child to 0. A process created by `exec(2)` inherits the time left on the old process's alarm clock.

RETURN VALUE

`Alarm` returns the amount of time remaining on the process's alarm clock from a previous call. If zero, no previous alarm was set.

DIAGNOSTICS

None.

SEE ALSO

`getitimer(2)`, `pause(2)`, `setitimer(2)`, `signal(2)`, `sigpause(2)`, `sigset(2)`.

NAME

async_daemon – start a BIOD server for asynchronous I/O requests

SYNOPSIS

```
int  async_daemon ()
```

DESCRIPTION

This system call does not normally return; instead, the process becomes a system process that asynchronously transfers blocks between memory and I/O devices. Normally, some number of BIOD processes are created when the system is brought to init level 1 or higher.

ACCESS CONTROL

Must be superuser to execute.

RETURN VALUE

−1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EPERM	The process has no permission to make the call.
EINTR	The process was terminated by a signal.

SEE ALSO

biod(1M).

NAME

`berk_sigpause` – set blocked signals and suspend process until a signal is caught

SYNOPSIS

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

int berk_sigpause (signal_mask)
int signal_mask;
```

where:

signal_mask Set of signals to be blocked while waiting

DESCRIPTION

`Berk_sigpause` assigns the set of signals specified in *signal_mask* to the set of signals blocked from presentation and then suspends the calling process until it is presented with a signal that is set to be caught. Changing the signal mask may cause previously pended signals to be presented immediately.

Neither the presentation of signals that are ignored, nor the presentation of signals that cause the termination of the calling process, nor the existence of pended signals cause `berk_sigpause` to return.

When a signal is caught by the calling process and control is returned from the signal handler, `berk_sigpause` returns. On return, the previous set of signals blocked from presentation is restored.

Signal "s" is represented by the value "SIGBIT(s)" in *signal_mask*.

It is not possible to block SIGKILL, SIGSTOP, or SIGCONT. It may or may not be possible to block signals that are not defined by the system. An attempt to block these signals will not produce an error.

ACCESS CONTROL

None.

RETURN VALUE

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EINTR A signal interrupted the `berk_sigpause` operation.

SEE ALSO

`sigblock(2), sigpause(2), sigvec(2).`

NAME

`bind` – bind a name to a socket

SYNOPSIS

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

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

where:

s	Socket to bind
name	Name to bind to socket
namelen	Length of name (bytes)

DESCRIPTION

`Bind` requests that address *name* be bound to socket *s*.

The rules for name binding vary between communication domains. Consult the related documentation for a specific domain for details about that domain.

Binding a name in the UNIX domain creates a file of type S_IFSOCK (socket special) in the file system that the caller must delete when it is no longer needed (using `unlink`). The file created is a side-effect of the current implementation, and may not be created in future versions of the UNIX IPC domain.

ACCESS CONTROL

None. (See related domain specific information for restrictions on names.)

RETURN VALUE

0	The call was successful.
–1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	*s* is not an active valid descriptor.
ENOTSOCK	*s* is not a socket.
EADDRNOTAVAIL	The address is not a valid address for the local machine.
EADDRINUSE	The address is already in use.
EINVAL	The socket is already bound to an address.
EACCES	The requested address is protected, and the current user has inadequate privilege to access it. Privilege is determined by the euid of the process when the socket was created.
EFAULT	The *name* parameter is not in a valid part of the user address space.
ENOBUFS	No internal buffers available.
EISCONN	Socket is already connected.

SEE ALSO

`connect`(2), `listen`(2), `select`(2), `socket`(2), `inet`(3N), `inet`(6F), `unix_ipc`(6F).

NAME

`brk` – change data segment space allocation

SYNOPSIS

```
#include <unistd.h>

int brk(void *endds);
```

where:

endds The address of the first byte beyond the new end of the data area

DESCRIPTION

The `brk()` system call dynamically changes 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 or deallocating the appropriate amount of space. The break value is the address of the first byte beyond the end of the data segment. The amount of allocated space increases as the break value increases.

If *endds* is greater than the current break value, any newly allocated pages will be initialized with zero bytes; i.e., if these bytes are read before they are written, the contents will be zero. If *endds* is less than the current break value, space is deallocated from the data segment. The contents of addresses from *endds* to the prior break value become undefined.

There is a maximum possible break value for a process; this value may be obtained by calling the `ulimit`(2) function. There is also a program-dependent minimum break value for a process; this minimum is greater than or equal to the address of the first byte in the data segment, and less than or equal to the program's initial break value.

The `brk()` system call will fail without making any change in the allocated space if an error occurs.

ACCESS CONTROL

No access check is made.

RETURN VALUE

Upon successful completion, `brk()` returns a value of `0`. Otherwise, it returns the value `-1`, and sets `errno` to indicate an error.

DIAGNOSTICS

Under the following conditions, `brk()` fails and sets `errno` to:

ENOMEM	if the change would allocate more space than is allowed by a system-imposed maximum (see `ulimit`(2)).
ENOMEM	if the change would allocate more space than is allowed by the current data resource limit (see `getrlimit`(2)).
ENOMEM	if the change would make the break value greater than or equal to the start address of an attached shared memory segment (see `shmat`(2)).
EFAULT	if the change would make the break value less than the system-imposed minimum.
EAGAIN	if the change would allocate more space than the available physical memory and swap space.
EAGAIN	if the `MCL_FUTURE` memory locking option is in effect for the calling process (see `memcntl`(2)), and the system-imposed limit on space locked into physical memory would be exceeded.

SEE ALSO

exec(2), getrlimit(2), memcntl(2), ulimit(2).

NAME

`chdir` – change the working directory of the calling process

SYNOPSIS

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

where:

path Address of a pathname

DESCRIPTION

Path points to a pathname naming a directory that is made the current working directory of the calling process. If *path* refers to a symbolic link, the target of the symbolic link is made the current working directory. The current working directory is the starting point of subsequent searches for pathnames that do not begin with '/'.

If the call fails, the current working directory is not changed.

ACCESS CONTROL

The calling process must have execute permission to the named directory.

The process must have permission to resolve *path*.

RETURN VALUE

0	The current directory was successfully changed.
–1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EACCES	Execute permission to the directory is denied.
ENOTDIR	The named file is not a directory.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

`chroot(2)`.

NAME

chmod – change mode of file

SYNOPSIS

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

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

where:

path	Address of a pathname
mode	File's new mode

DESCRIPTION

The chmod system call changes the mode (including permissions) associated with a file. *path* points to a pathname naming a file of type ordinary, directory, FIFO, block special, character special, or symbolic link. If *path* refers to a symbolic link, the target of the symbolic link is handled, not the symbolic link. The file must reside on a file system device mounted read-write. Chmod sets the protection rights, sticky bit, set-user-id bit, and set-group-id bit of the file's mode according to *mode*.

Values of *mode* are constructed by joining one or more of the following flags with a logical OR:

S_ISUID (04000)	Set user id on execution.
S_ISGID (02000)	Set group id on execution. If the (S_IEXEC >> 3) bit is not set and the file is an ordinary file, this bit enables mandatory record locking for the file.
S_ISVTX (01000)	Sticky bit. Some versions of the UNIX™ system attempt to optimize access to executable files (that have this bit set) by maintaining a copy of the program image in a memory- or disk-based file system cache. The DG/UX system attempts this optimization for all executable images. For files of type 'directory' and 'control point directory', the sticky bit has a further meaning. If the sticky bit is set, then the directory is considered append only. Processes without appropriate permissions cannot delete or rename files owned by other users in such a directory.
S_IREAD (00400)	Read by owner.
S_IWRITE (00200)	Write by owner.
S_IEXEC (00100)	Execute (search, if a directory) by owner.
(S_IREAD >> 3) (00040)	Read by group.
(S_IWRITE >> 3) (00020)	Write by group.
(S_IEXEC >> 3) (00010)	Execute (search) by group.
(S_IREAD >> 6) (00004)	Read by others.
(S_IWRITE >> 6) (00002)	Write by others.
(S_IEXEC >> 6) (00001)	Execute (search) by others.

For each flag set in *mode,* the corresponding attribute bit or protection right is set. The other attribute bits and protection rights are cleared. If the calling process attempts to set the sticky bit or the set-group-id bit but does not meet the requirements for doing so (see access control), that bit is cleared, but the process is not notified of the failed attempt. One of the access requirements to perform this call (the effective user id of the process must be superuser or match the file's user id) coincides with the access needed to set the set-user-id bit, hence the process may always set that bit if it chooses.

The time of last change to the file's attributes is set to the current time.

If `chmod` fails, the file's attributes remain unchanged.

ACCESS CONTROL

The effective user id of the calling process must be superuser or match the user id of the file.

The process's effective user id must be superuser to set the sticky bit.

To set the set-group-id bit, either

- the process's effective user id must be superuser,
- the process's effective user id must match the user id of the file and the process's effective group id must match the file's group id.

Failure to meet the requirements for setting one of these bits does not produce an error. Note that meeting the first access requirement is sufficient to allow a process to set the set-user-id bit.

The process must have permission to resolve *path.*

RETURN VALUE

0	The file's mode was successfully changed.
−1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EROFS	The named file resides on a file system device mounted read-only.
EPERM	The file's user id does not match yours, and you are not the superuser.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.

EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space, or the pathname does not terminate in the process's address space.

SEE ALSO

chmod(1), chown(2), creat(2), fchmod(2), fchown(2), fcntl(2), fstat(2), mknod(2), open(2), read(2), stat(2), write(2).

NAME

chown, lchown – change user id and group id of a file

SYNOPSIS

```
#include <unistd.h>

int  chown (path, user, group)
char * path;
int  user;
int  group;

int  lchown (path, user, group)
char * path;
int  user;
int  group;
```

where:

path	Address of a pathname
user	File's new user id
group	File's new group id

DESCRIPTION

Chown sets the file's user id (st_uid) and group id (st_gid) to the numeric values *user* and *group*, respectively. *path* points to a pathname naming a file of type ordinary, directory, FIFO, block special, character special, or symbolic link. The file cannot reside on a file system device mounted read-only.

If the value of *user* is –1, the user id of the file is left unchanged. Similarly, if the value of *group* is –1, the group id of the file is left unchanged.

The set-user-id and set-group-id bits of the file mode (st_mode) are left unchanged unless the effective user id of the calling process is not superuser, in which case they are cleared.

The file's time of last attribute change (st_ctime) is set to the current time.

If chown fails, the user id, group id, and attributes of the file remain unchanged.

Chown and lchown operate identically except for their handling of symbolic links. If the call is to lchown and *path* refers to a symbolic link, the symbolic link is handled, not the target of the symbolic link. chown will handle the target of the symbolic link.

ACCESS CONTROL

The effective user id of the calling process must be superuser or match the user id of the file.

The process must have permission to resolve *path*.

RETURN VALUE

0	The user id and group id of the file were successfully changed.
–1	An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EPERM	Permission to change the file's user and group id is denied.
EROFS	The named file resides on a file system device mounted read-only.

ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

chmod(1), fchmod(2), fchown(2).

NAME

`chroot` – change the root directory of the calling process

SYNOPSIS

```
#include <unistd.h>

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

where:

path Address of a pathname

DESCRIPTION

Path points to the pathname of a directory. `chroot` makes that directory the root directory of the calling process, the starting point of searches for pathnames beginning with '/'. If *path* refers to a symbolic link, the target of the symbolic link is made the root directory. The process's working directory is unaffected by the `chroot` system call.

The '..' entry in the root directory means the root directory itself; the root is treated as having no parent directory. Thus, the process cannot access files outside the subtree whose topmost node is the root directory, unless the process's current working directory is located outside of that subtree.

If `chroot` fails, the root directory remains unchanged.

ACCESS CONTROL

The effective user id of the calling process must be superuser.

RETURN VALUE

0 The root was successfully changed.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

ENOENT	The named directory does not exist.
EPERM	Permission to change the root directory is denied.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the

process's address space.

SEE ALSO

chdir(2).

NAME

close – close an object associated with a file descriptor

SYNOPSIS

```
int close (fildes)
int fildes;
```

where:

fildes A valid, active file descriptor

DESCRIPTION

If *fildes* is a valid, active descriptor, close breaks the connection between the descriptor indicated by *fildes* and the object to which it refers. If *fildes* is the last descriptor that refers to an object pointer, the object pointer is "closed" by invoking the type-specific close operation.

Thus, the type manager is informed only of the last close operation of an object pointer. These managers (and specifically, the device driver close operations) should not expect to be invoked during each close system call.

Upon completion of the close operation, *fildes* will be inactive. Until *fildes* is reallocated, subsequent operations using *fildes* will result in an EBADF error condition.

If a STREAMS-based *fildes* is closed, and the calling process had previously registered to receive a SIGPOLL signal for events associated with that stream in, the calling process will be unregistered for events associated with the stream. The last close for a stream causes the stream associated with *fildes* to be dismantled. If O_NDELAY and O_NONBLOCK are clear and there have been no signals posted for the stream, and if there are data on the module's write queue, close waits up to 15 seconds (for each module and driver) for any output to drain before dismantling the stream. The time delay can be changed via an I_SETCLTIME ioctl. If O_NDELAY or O_NONBLOCK is set, or if there are any pending signals, close does not wait for output to drain, and dismantles the stream immediately.

If *fildes* is associated with one end of a pipe, the last close causes a hangup to occur on the other end of the pipe. In addition, if the other end of the pipe has been named [see fattach(3C)], the last close forces the named end to be detached [see fdetach(3C)]. If the named end has no open processes associated with it and becomes detached, the stream associated with that end is also dismantled.

Objects are also subject to implicit close operations via the exit and exec operations. When a process terminates, all active descriptors are closed. When a process performs a successful exec operation, all active descriptors with the 'close-on-exec' attribute are closed. These implicit close operations are equivalent to an explicit close operation.

On each close of an object, all outstanding locks owned by the calling process on the object are released.

ACCESS CONTROL

None.

RETURN VALUE

0 The object was successfully closed.

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EBADF	*fildes* is not a valid, active descriptor.
EINTR	Close call was interrupted.
EIO	An i/o error occurred while reading from or writing to the file system. The *fildes* is inactived.

SEE ALSO

`accept`(2), `creat`(2), `dup`(2), `exec`(2), `fcntl`(2), `open`(2), `pipe`(2), `signal`(2), `sigset`(2), `socket`(2), `socketpair`(2).

NAME

connect – initiate a connection on a socket

SYNOPSIS

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

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

where:

s The file descriptor of a socket to connect

name Name of peer or listening socket through which the connection will be made

namelen Length of name (bytes)

DESCRIPTION

The parameter *s* is a socket. If it is of type SOCK_DGRAM, then this call specifies the peer to which datagrams are to be sent; if it is of type SOCK_STREAM, then this call tries to make a connection through a listening socket specified by *name*, which is an address in the communications space of the socket.

ACCESS CONTROL

None. See the related documentation on the individual communication protocol for specific domain interpretations.

RETURN VALUE

0 Completed successfully, a connection has been established.

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EBADF	*s* is not active, valid descriptor.
ENOTSOCK	*s* is a descriptor for a file, not a socket.
EADDRNOTAVAIL	The specified address is not available on the specified host.
EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.
EISCONN	The socket is already connected.
ETIMEDOUT	Connection establishment timed out without establishing a connection.
ECONNREFUSED	The attempt to connect was rejected by foreign host.
ENETUNREACH	The network isn't reachable from this host.
EADDRINUSE	The address is already in use.
EFAULT	The *name* parameter specifies an area outside the process address space.
EAGAIN	The socket is non-blocking and the connection cannot be completed immediately.
ENOBUFS	No internal buffers available.

EINVAL	Invalid system call argument (probably name length).
EALREADY	The connect operation has already been started on this socket and has not yet finished. (An earlier call must have returned EAGAIN or EINTR.)
EINTR	System call returned due to interrupt.
EOPNOTSUPP	The socket is in the listen state.

SEE ALSO

`accept`(2), `listen`(2), `select`(2), `socket`(2), `getsockname`(2).

NAME

creat – create a new file or rewrite an existing one

SYNOPSIS

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

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

where:

path Address of a pathname

mode Protection mode of the new file

DESCRIPTION

Creat(2) is equivalent to:

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

See open(2) for more details.

ACCESS CONTROL

Same as for the open system call.

RETURN VALUE

See open(2).

DIAGNOSTICS

See open(2).

SEE ALSO

chmod(2), close(2), dup(2), fcntl(2), lseek(2), open(2), read(2), umask(2), write(2), stat(5).

NAME

`dg_allow_shared_descriptor_attach` – let processes attach shared descriptor array

SYNOPSIS

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

int  dg_allow_shared_descriptor_attach (pid_t pid)
```

where:

pid The process identifier of the process to be given permission to attach.

DESCRIPTION

Allow the process identified by *pid* to attach to the shared descriptor array of the calling process. This will enable process *pid* to do a successful `dg_attach_to_shared_descriptors`(2) on the current process.

The right to attach to a shared descriptor array is inherited across forks and execs of process *pid*.

ACCESS CONTROL

No access checking is performed.

RETURN VALUE

0 Process *pid* may now attach to the shared descriptor array of the calling process.

-1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

ESRCH *pid* does not exist.

SEE ALSO

`open`(2), `dg_attach_to_shared_descriptors`(2).

NAME

dg_attach_to_shared_descriptors – attach another process's shared descriptor array

SYNOPSIS

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

int  dg_attach_to_shared_descriptors (pid_t pid)
```

where:

pid The process identifier of the process whose shared descriptor array is to be attached.

DESCRIPTION

File descriptors fall into two classes based on their process access and permanance semantics. The first class of file descriptor is a per-process file descriptor. A per-process descriptor is accessible only from the current process. A per-process descriptor is closed only as a by-product of some action taken by the current process. This class of file descriptor is never shared by other processes.

The second class of file descriptor is a shared descriptor. Shared descriptors are collected into a shared descriptor array, which is the granularity upon which any process sharing of descriptors is done. The shared descriptor array and all shared descriptors in that array persist only as long as the shared descriptor array is attached to at least one process. If a shared descriptor array is no longer referenced by any process then it will be destroyed and all remaining descriptors in the array will be closed. References to the shared descriptor array are lost either when a process exits and when it attaches to another shared descriptor array.

A shared descriptor is accessible from all processes that have attached to the same descriptor array. A shared descriptor may be closed by any process to which it is attached. `Dg_attach_to_shared_descriptors`(2) attaches the shared process array of process *pid* to the calling process. The attach operation will fail if the attach would cause the per-process soft limit on the maximum number of descriptors to be exceeded. When the attach is completed, the shared descriptor arrary (if any) previously attached to the calling process is no longer attached (the individual descriptors may or may not be closed) and the shared descriptor array of *pid* is now attached to the calling process.

Shared descriptors should be used only by processes that are prepared to cooperate in their use. Since shared descriptors may be closed by any process that have access to it, process must be prepared to lose access to a descriptor because of the action of another process.

ACCESS CONTROL

The process identified by *pid* must have previously issued a successful `dg_allow_shared_descriptor_attach`(2) on the pid of the calling process for this call to be successful.

RETURN VALUE

0 All descriptors in the shared descriptor array of process *pid* may now be accessed by the calling process.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

ESRCH	*pid* does not exist.
EMFILE	The attach operation would exceed the soft limit on the number of descriptors per process.
EPERM	The calling process does not have permission to attach to the shared descriptor array of process *pid*.

SEE ALSO

`open`(2), `dg_allow_shared_descriptor_attach`(2).

NAME

`dg_decryptsessionkey` – decrypt conversation key with the client/server common key

SYNOPSIS

```
int  dg_decryptsessionkey (netname, deskey)
char * netname;
des_block * deskey;
```

where:

netname Netname of the server

deskey Pointer to the DES key to decrypt

DESCRIPTION

This call is used to request the user keyserver process to decrypt a conversation key with the common key for this user and the server machine.

ACCESS CONTROL

None.

RETURN VALUE

0 The operation was successful.

–1 An error occurred. `errno` indicates the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EINVAL	Secure RPC is not configured. Secure RPC using DES Authentication is an additional feature that must be purchased separately from the DG/UX™ ONC™/NFS® product.
ENOMEM	Kernel memory could not be allocated to read in the *netname*.
EFAULT	Some part of the string pointed to by *netname* lies outside the process's readable address space.
EFAULT	Some part of the string pointed to by *deskey* lies outside the process's writable address space.

SEE ALSO

`dg_encryptsessionkey(2)`, `dg_getrootkey(2)`, `dg_setsecretkey(2)`.

NAME

`dg_devctl` – perform device-control functions

SYNOPSIS

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

int      dg_devctl (cmd, arg)
unsigned int cmd;
void     * arg;
```

where:

cmd One of the following command names: DG_DEVCTL_CONFIGURE_DEVICE, DG_DEVCTL_DECONFIGURE_DEVICE, DG_DEVCTL_NAME_TO_DEVICE, DG_DEVCTL_DEVICE_TO_NAME

arg Pointer to a packet of information used and/or filled in by the command

DESCRIPTION

`Dg_devctl`(2) can be used to perform a variety of device-related tasks on the system. The specific task to be executed is indicated by the *cmd* parameter, and the address of an information packet used to store information for that command is passed in the *arg* parameter. The various command values, and the types of their accompanying argument packets, are defined and described in `<sys/dg_devctl.h>`.

The DG_DEVCTL_CONFIGURE_DEVICE command is used to configure a device into the system, given only its name in DG/UX common device specification format.

The DG_DEVCTL_DECONFIGURE_DEVICE command is used to deconfigure a device out of the system, given only its name in DG/UX common device specification format.

The DG_DEVCTL_NAME_TO_DEVICE command is used to find out the device number of a device, given only its name in DG/UX common device specification format.

The DG_DEVCTL_DEVICE_TO_NAME command is used to find out the canonical DG/UX common device specification format name of a device, given only its device number.

ACCESS CONTROL

Any user may execute the DG_DEVCTL_NAME_TO_DEVICE and DG_DEVCTL_DEVICE_TO_NAME commands.

Only the superuser may execute the DG_DEVCTL_CONFIGURE_DEVICE and DG_DEVCTL_DECONFIGURE_DEVICE commands.

RETURN VALUE

0 The `dg_devctl` operation was successful.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EPERM A process called `dg_devctl()` without having an effective user ID of 0.

EINVAL *cmd* is not one of the valid commands described above.

EFAULT *arg* points to an invalid address.

ENXIO	An attempt was made to configure a device that was already configured.
EBUSY	An attempt was made to deconfigure a busy or undeconfigurable device.
ENXIO	An attempt was made to deconfigure, get the name, or get the device number of a device that is not configured.
ENXIO	An attempt to configure or deconfigure a device failed for an unknown reason.
EINVAL	An attempt was made to get the name of a device, but not enough string storage was allocated to receive the name.

SEE ALSO

diskman(1M), dg_sysctl(2).

NOTE

This system call exists only for backwards compatibility with prior versions of DG/UX. It will be removed in a future revision. Use the dg_sysctl(2) system call instead.

NAME

`dg_encryptsessionkey` – encrypt conversation key with the client/server common key

SYNOPSIS

```
int dg_encryptsessionkey (netname, deskey)
char * netname;
des_block * deskey;
```

where:

netname	Netname of the server
deskey	Pointer to the des key to encrypt

DESCRIPTION

This call is used to request the user keyserver process to encrypt a conversation key with the common key for this user and the server machine.

ACCESS CONTROL

None.

RETURN VALUE

0	The operation was successful.
–1	An error occurred. `errno` indicates the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EINVAL	Secure RPC is not configured. Secure RPC using DES Authentication is an additional feature that must be purchased separately from the DG/UX™ ONC™/NFS® product.
ENOMEM	Kernel memory could not be allocated to read in the *netname*.
EFAULT	Some part of the string pointed to by *netname* lies outside the process's readable address space.
EFAULT	Some part of the string pointed to by *deskey* lies outside the process's writable address space.

SEE ALSO

`dg_decryptsessionkey(2), dg_getrootkey(2), dg_setsecretkey(2).`

NAME

`dg_ext_errno` – return the extended `errno` for the current process

SYNOPSIS

`long dg_ext_errno ()`

DESCRIPTION

This function returns the extended `errno` which is set on return from a system call. The high order word contains the subsystem-id of the extended `errno`. The low order word contains the specific error in the specified subsystem. The high order bit of the extended `errno` is always set.

RETURN VALUE

extended_errno   Returns the extended_errno.

DIAGNOSTICS

None.

SEE ALSO

`perror`(3C).

NAME

dg_file_info – get file usage information for process identified by process key

SYNOPSIS

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

int     dg_file_info (process_key,
        descriptor_ptr,
        file_info_buffer_size_ptr,
        file_info_buffer_ptr,
        version)
long    process_key;
long    * descriptor_ptr;
long    * file_info_buffer_size_ptr;
struct dg_file_info * file_info_buffer_ptr;
long    version;
```

where:

process_key	The key obtained from a previous dg_process_info call; used to identify the process of interest
descriptor_ptr	The type of file usage information to return
file_info_buffer_size_ptr	On input, the maximum number of "file info" structures to be returned; on output, the number of "file info" structures returned
file_info_buffer_ptr	A pointer to an area of least **file_info_buffer_size_ptr* *sizeof (struct dg_file_info) bytes. The area you pass a pointer to can be as large as you want. Information about files is put here.
version	DG_FILE_INFO_VERSION. If *version* is not this, no information will be returned.

DESCRIPTION

Version specifies the version of information the user is interested in. DG_FILE_INFO_VERSION always means the "current version". *Process_key* indicates a process whose file information is to be returned. **descriptor_ptr* indicates the starting point for a linear search through the following objects:

- Process's descriptor table
- Process's current working directory
- Process's current root directory
- File currently being executed by the process.

Information about up to **file_info_buffer_size_ptr* entities is returned in the area indicated by *file_info_buffer_ptr*. **file_info_buffer_size_ptr* is set to indicate the number of entities for which information is returned.

Always, information about the first **file_info_buffer_size_ptr* active entities is returned.

The data is put into the buffer as a series of dg_file_info structures. The "file_info" structure is defined as:

```
struct dg_file_info {
      int      version;
```

```
            int      descriptor;
            struct stat stat_pkt;
    };
```

The *version* field indicates the version and, therefore, the format of the information that follows. The *descriptor* field indicates the object the stat information refers to. If *descriptor* is DG_FILE_INFO_CWD, the `stat` information refers to the process's current working directory. If *descriptor* is DG_FILE_INFO_ROOT_DIR, the `stat` information refers to the process's current root directory. If *descriptor* is DG_FILE_INFO_COMMAND, the `stat` information refers to the file currently being executed by the process. Otherwise, *descriptor* is a descriptor currently active in the process indicated by *process_key*. Only the first **file_buffer_size_ptr* entries contain valid information.

Upon return, the actual number of file_info structures put into the area pointed to by *file_info_buffer_ptr* is returned in **file_info_buffer_size_ptr*. **descriptor_ptr* is set to the starting point for the next `dg_file_info` call. If **descriptor_ptr* is −1, there are no more entities for which information can be returned.

ACCESS CONTROL

No access control is provided.

RETURN VALUE

0 Successful completion.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EFAULT Either *file_info_buffer_size_ptr*, *descriptor_ptr* or *file_info_buffer_ptr* points to an invalid address.

EINVAL *version* requested is not supported.

EINVAL **descriptor_ptr* is not one of the following: DG_FILE_INFO_CWD, DG_FILE_INFO_ROOT_DIR, DG_FILE_INFO_COMMAND, or 0 through NOFILE−1 inclusive.

SEE ALSO

`dg_process_info`(2).

NAME

`dg_fstat` – get extended file status information

SYNOPSIS

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

int    dg_fstat (fildes, buffer_ptr, version)
int    fildes;
struct dg_stat * buffer_ptr;
unsigned short version;
```

where:

fildes	A valid, active file descriptor
buffer_ptr	Address of a `dg_stat` buffer to fill
version	Version of the `struct dg_stat` packet that *buffer_ptr* refers to; should be set to DG_STAT_VERSION_NUMBER

DESCRIPTION

`Dg_fstat`(2) returns the current extended attributes of the file referenced by *fildes* into the `dg_stat` buffer at the location specified by *buffer_ptr*. If `dg_fstat` fails, the contents of the buffer are undefined.

The size and composition of the structure referred to by *buffer_ptr* is determined by the *version* parameter. All calls to this function should use DG_STAT_VERSION_NUMBER for this parameter. *version* allows for future revisions of struct `dg_stat` to be handled in a compatible way.

The interpretation of the file's attributes depends on the file's type (see `dg_stat`(5) and `stat`(5)).

ACCESS CONTROL

Read, write, or execute permission of the open file is not required. However, for *fildes* to be active, the file must be open for reading or writing.

RETURN VALUE

0 The `dg_fstat` operation was successful.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EINVAL	*version* is not a supported version of struct dg_stat.
EFAULT	*buffer_ptr* points to an invalid address.
EBADF	*Fildes* is not a valid, active file descriptor.

SEE ALSO

chmod(2), **chown**(2), **creat**(2), **dg_mstat**(2), **dg_stat**(2), **fchmod**(2), **fchown**(2), **fstat**(2), **link**(2), **lstat**(2), **mknod**(2), **pipe**(2), **read**(2), **stat**(2), **time**(2), **unlink**(2), **utime**(2), **utimes**(2), **write**(2), **dg_stat**(5), **stat**(5).

NAME

dg_getrootkey – get root's secret key

SYNOPSIS

```
int dg_getrootkey (secretkey)
char * secretkey;
```

where:

secretkey The root secret key.

DESCRIPTION

This call is used to read the root's decrypted secret key from battery backed-up RAM. It is used by the keyserv(8C) process to initialize its database. In this way, the keyserver can get the root key without operator intervention, as in the case of a power failure in the middle of the night.

ACCESS-CONTROL

None.

RETURN VALUE

0 The operation was successful.

-1 An error occurred. errno indicates the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EPERM The process's effective user id is not superuser.

EFAULT Some part of the string pointed to by *secretkey* lies outside the process's writable address space.

SEE ALSO

dg_decryptsessionkey(2), dg_encryptsessionkey(2), dg_setsecretkey(2), keyserv(8C).

NAME

dg_ipc_info – get information about current IPCs state

SYNOPSIS

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

int    dg_ipc_info(ipc_component, buffer_ptr, key_ptr, version)
int    ipc_component;
char   * buffer_ptr;
key_t  * key_ptr;
long   version;
```

where:

ipc_component The IPC component: IPC_MSQ (message queues), IPC_SEM (semaphores), or IPC_SHM (shared memory).

buffer_ptr A pointer to a user buffer for returned information. The buffer should be of struct msqid_ds, semid_ds or shmid_ds type, depending on the value of *ipc_component*.

key_ptr The value of **key_ptr* should be set to DG_IPC_INFO_INITIAL_KEY on the first call for each IPC component. On return, **key_ptr* contains a value to which **key_ptr* should be assigned on a subsequent call to dg_ipc_info.

version The version of this call (the most recent version is DG_IPC_INFO_CURRENT_VERSION). See dg_ipc_info.h.

DESCRIPTION

This is an alternative interface to /dev/kmem. This system call returns information about the current state of the IPC components–shared memory, message queues, and semaphores, as selected by *ipc_component*.

The dg_ipc_info call searches the *ipc_component* data structures. The first valid entry it finds is copied into the user buffer pointed to by *buffer_ptr*. **key_ptr* is assigned to the value that should be used by the next dg_ipc_info call to get the next valid entry of *ipc_component*. If no valid entry is found, the dg_ipc_info call returns with errno set to an appropriate value.

RETURN VALUE

1 This value indicates a successful return of information about an IPC structure, but more structures may still be available.

0 Successful completion. This means that there are no more IPC structures of type *ipc_component* to return. The contents of *buffer* are undefined.

–1 Error. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EINVAL Invalid argument–*ipc_component* is not IPC_SHM, IPC_MSQ, or IPC_SEM.

EINVAL Invalid argument–*version* is not a valid version number.

EINVAL Invalid argument–*key_ptr* points to an invalid index key.

EFAULT *buffer_ptr* or *key_ptr* is an invalid address.

SEE ALSO

exec(2), ulimit(2).

NAME

`dg_lcntl` – process a record lock request on a filehandle

SYNOPSIS

```
#include <sys/fcntl.h>
#include <sys/nfs.h>

int dg_lcntl(cmd, file_ptr, client_id, client_id_inuse_ptr, flock_ptr)
int             cmd;
fhandle_t *     file_ptr;
int             client_id;
int *           client_id_inuse_ptr;
struct flock *  flock_ptr;
```

where:

cmd	A lock command
file_ptr	An nfs file handle
client_id	A client id for the lock
client_id_inuse_ptr	Whether lock client id is in use
flock_ptr	Lock parameters

DESCRIPTION

`Dg_lcntl` processes the *cmd* lock request on the file given by the file handle *file_ptr* and lock client identifier *client_id* with the lock parameters given in *flock_ptr*. Upon return, the value of *client_id_inuse_ptr* equals 1 if the *client_id* currently holds any locks or lock requests, otherwise the value of *client_id_inuse_ptr* equals 0.

`Dg_lcntl` provides a variety of lock operations on file handles. It is similar to `fcntl`(2), but takes a file handle argument rather than a file descriptor, and a lock client identifier rather than using the process id of the caller. A lock client id is a small integer specified by the caller. *Cmd* is the lock command to be performed on *file_ptr* with *client_id* and *flock_ptr* specifying the lock parameters. The *flock_ptr* parameters are treated the same as the `fcntl`(2) lock parameters. On return, *client_id_inuse_ptr* specifies whether *client_id* holds any locks or pending lock requests. The commands available are:

DG_LCNTL_SETLK
: Set or clear a file lock according to *flock_ptr*. DG_LCNTL_SETLK is used to set read and write locks, or remove either type of lock. If a read or a write lock cannot be set, `dg_lcntl` returns immediately with the value –1 and `errno` set to EACCES. [See the `fcntl`(2) command F_SETLK.]

DG_LCNTL_SETLKD
: This command is the same as DG_LCNTL_SETLK except if a read or write lock is blocked by other locks, `dg_lcntl` queues a delayed lock request and returns with the value –1, and `errno` set to EINPROGRESS. The lock request is attempted when the blocking lock is released. The results of the delayed request are returned by the `dg_lock_wait`(2) call. If *client_id* already owns a delayed lock request, then `dg_lcntl` returns –1 and `errno` is set to ENOLOCK.

DG_LCNTL_GETLK
: Get the first lock that blocks the lock description specified by *flock_ptr*. The information retrieved overwrites the information passed to `dg_lcntl` in *flock_ptr*. If no lock is found that would prevent this lock from being set,

then *flock_ptr* is unchanged, except for the lock type, which is set to DG_LCNTL_UNLCK. [See the `fcntl(2)` command F_GETLK.]

DG_LCNTL_CANCEL
Remove the delayed lock request specified by *flock_ptr*. If there is no delayed lock request, then this command removes the lock specified by *flock_ptr*. This is the only command that may be issued for *client_id* while a delayed lock request exists for *client_id*.

DG_LCNTL_RECLAIM
Set or clear a file lock according to *flock_ptr*. DG_LCNTL_RECLAIM is used during the system restart grace period to reclaim read and write locks. If a read or a write lock cannot be set, `dg_lcntl` returns immediately with the value −1, and `errno` is set to ENOLINK.

The only process that uses this function is the network lock server, `rpc.lockd`.

ACCESS CONTROL

The caller must be super-user.

RETURN VALUE

0 The `dg_lcntl` operation was successful.

−1 An error occurred. `errno` indicates the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes regardless of the value of *cmd*:

EPERM
Must be super-user to use this system call.

EINVAL
cmd is not one of the known values.

EACCES
The *cmd* is DG_LCNTL_SETLK and the type of lock sought is a read lock (DG_LCNTL_RDLCK) or write lock (DG_LCNTL_WRLCK), and either the segment of a file to be locked is already write-locked by another process, or the type is a write lock and the segment of a file to be locked is already read-locked or write-locked by another process..

EFAULT
One of the arguments points outside of the process's readable address space.

ESTALE
The filehandle specified in the request is no longer valid.

EINTR
The process received a signal while processing the lock request.

EDEADLK
The *cmd* is DG_LCNTL_SETLKW or DG_LCNTL_SETLKD and a deadlock would exist if the lock were granted or allowed to pend.

EINPROGRESS
The *cmd* is DG_LCNTL_SETLKD and the lock request can not be granted immediately, but has been queued for later completion. When the lock currently blocking the request is released, the request will be retried. The results of the delayed request are returned by the `dg_lock_wait(2)` call.

ENOLCK
The *cmd* can not be satisfied because there are no more record locks available.

ENOLCK
The *cmd* is DG_LCNTL_SETLKD and the *client_id* already has a pending lock request.

ENOLCK — The *cmd* is DG_LCNTL_RECLAIM but the lock specified by *flock_ptr* can not be granted.

SEE ALSO

`fcntl`(2), `dg_lock_reset`(2), `dg_lock_wait`(2), `lockf`(3C), `fcntl`(5).

NAME

`dg_lock_kill` – remove locks held by remote lock clients

SYNOPSIS

```
#include <sys/fcntl.h>
#include <sys/nfs.h>

int   dg_lock_kill (count, client_id_list_ptr)
int   count;
int * client_id_list_ptr;
```

where:

count	Count of client id's in the list
client_id_list_ptr	A list of client id's to free

DESCRIPTION

Remove all locks and lock requests owned by the client in *client_list_ptr*. *count* gives the number of entries in the list.

The only process that uses this function is the network lock server, `rpc.lockd`.

ACCESS CONTROL

The caller must be super-user.

RETURN VALUE

0	The `dg_lock_kill` operation was successful.
−1	An error occurred. `errno` indicates the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EPERM	Must be super-user to use this system call.
EINVAL	Count is less than or equal to zero.
ENOMEM	There is not enough memory to process the request.
EFAULT	One of the arguments points outside of the process's readable address space.
EINTR	The process received a signal while processing the request.
ENOLOCK	The command can not be satisfied because there are no more record locks available.

SEE ALSO

fcntl(2), **dg_lcntl**(2), **dg_lock_reset**(2), **dg_lock_wait**(2), **lockf**(3C), **fcntl**(5).

NAME

dg_lock_reset – reset remote file lock database, start lock reclaim grace period

SYNOPSIS

```
int      dg_lock_reset (grace)
time_t   grace;
```

where:

grace The number of seconds in the grace period

DESCRIPTION

The dg_lock_reset system call removes all (remote) locks set by the network lock server. It pends all lock requests, and deny all remote lock requests, for *grace* seconds in order to allow remote clients to reclaim their locks.

The only process that uses this function is the network lock server, rpc.lockd.

ACCESS CONTROL

The caller must be super-user.

RETURN VALUE

0 The dg_lock_reset operation was successful.

–1 An error occurred. errno indicates the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EPERM Must be super-user to use this system call.

EINVAL The grace period is invalid.

SEE ALSO

dg_lcntl(2), dg_lock_wait(2), fcntl(2), lockf(3C), fcntl(5).

NAME

dg_lock_wait – wait for previously delayed lock requests to complete

SYNOPSIS

```
int    dg_lock_wait (client_id_ptr, client_id_in_use_ptr)
int * client_id_ptr;
int * client_id_in_use_ptr;
```

where:

client_id_ptr	Space to return the client id of the completed request
client_id_in_use_ptr	Space to return whether the lock client id is in use

DESCRIPTION

Dg_lock_wait(2) suspends the calling process until either a signal is received, or a previously issued lock request that has been delayed completes.

If a previously issued lock request completes, the *client_id_ptr* argument identifies the client of the completed request, and *client_id_in_use_ptr* indicates whether client currently holds any locks or lock requests. The return value may be 0 or −1, depending upon whether the request was successful.

If a signal is received before any requests complete, then dg_lock_wait returns and both *client_id_ptr* and *client_id_in_use_ptr* are invalid. In this case, the return value is −1, and errno is set to EINTR.

The only process that uses this function is the network lock server, rpc.lockd.

ACCESS CONTROL

The caller must be super-user.

RETURN VALUE

0 The dg_lock_wait operation was successful. The *client_id_ptr* lock request was granted.

−1 An error occurred. errno indicates the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EPERM	Must be super-user to use this system call. The content of both *client_id_ptr* and *client_id_in_use_ptr* are invalid.
EFAULT	One of the arguments points outside the process's readable address space. The content of both *client_id_ptr* and *client_id_in_use_ptr* are invalid.
EDEADLK	The *client_id_ptr* lock request is refused because it would cause a deadlock. Both *client_id_ptr* and *client_id_in_use_ptr* are valid.
EINTR	A signal was received. The content of both *client_id_ptr* and *client_id_in_use_ptr* are invalid.

SEE ALSO

dg_lcntl(2), dg_lock_reset(2), fcntl(2), lockf(3C), fcntl(5).

NAME

dg_mknod – create a file system node

SYNOPSIS

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

int    dg_mknod (path, buffer_ptr, version)
char   * path;
struct dg_mknod * buffer_ptr;
unsigned short  version;
```

where:

path	Address of pathname to create
buffer_ptr	Address of a dg_mknod buffer which describes the node to be created
version	Version of the struct dg_mknod packet that *buffer_ptr* refers to; should be set to DG_MKNOD_VERSION_NUMBER

DESCRIPTION

Dg_mknod(2) creates a new ordinary file, directory, control-point directory, block-special file, character-special file, FIFO file, or symbolic link file. The new file will be named *path*, and its attributes will be set according to the struct dg_mknod packet represented by *buffer_ptr*:

- The file's type and mode will be set according to the *extended_mode* field. Note that the file's mode is modified by the process's file mode creation mask; all bits set in the mask are cleared (see umask(2)). Note also that only the superuser may set the sticky bit (S_ISVTX), as explained below.
- If the file is of type block-special (S_IFBLK) or character-special (S_IFCHR), then the file's represented device (st_rdev) will be set to *device_number*.
- If the file is of type FIFO (S_IFIFO), then the indicated FIFO (named pipe) file will be created.
- If the file is of type symbolic link (S_IFLNK), then the pathname denoted by the *symbolic_link_target* field will be used as the target of the link file. Note that there is no requirement that *symbolic_link_target* actually exist.
- If the file is of type ordinary file (S_IFREG), directory (S_IFDIR) or CPD (DG_IFCPD), then the file's data and index element sizes will be set according to the information in *buffer_ptr*, using the following algorithm: Each integer between *desired_data_element_blocks* and *data_element_blocks_limit*, starting at the former, will be examined in order. The first number that is discovered to be a valid data element size is the number that will be used as the data element size. If no number in the specified range is a valid element size, an error will be returned (see below) and no node will be created. The file's index element size will be set in exactly the same manner, except that the range will start at *desired_index_element_blocks* and work towards *index_element_blocks_limit*.
- If the file is of type socket (S_IFSOCK), or if the file type is invalid, an error will be returned (see below) and no node will be created.

The file's other attributes are initialized as follows:

- The file's inode number (st_ino) is set to refer to the per-file database allocated.
- The file's size (st_size) is set to zero.
- The number of links to the file (st_nlink) is set to one, unless the file is of type directory (S_IFDIR) or CPD (DG_IFCPD), in which case it is set to two.
- The file's user-ID (st_uid) is set to the effective user-ID of the calling process.
- The file's group-ID (st_gid) is set to the effective group-ID of the calling process.
- The file's time fields (st_atime, st_ctime and st_mtime) are all set to the current time.

Path is created in the containing directory and is made to identify the newly created file. The attributes of the parent directory change as follows:

- The file size (st_size) is updated if the new directory entry caused the directory to change size.
- The time last modified (st_mtime) and time of last attribute change (st_ctime) are set to the current time.

If the call to dg_mknod() fails, no file is created, and the attributes of the directory intended to contain the file remain unchanged.

The size and composition of the structure referred to by *buffer_ptr* are determined by the *version* parameter. All calls to this function should use DG_MKNOD_VERSION_NUMBER for this parameter. *Version* allows for future revisions of struct dg_mknod to be handled in a compatible way.

ACCESS CONTROL

The process must have write access to the containing directory of *path*, and it must have permission to resolve *path*.

The process's effective user-ID must be superuser in order to create files of type block-special or character-special.

The process's effective user-ID must be superuser in order to set the sticky-bit (S_ISVTX). However, failure to meet this requirement will not produce an error when setting the sticky bit is requested; the file will merely be created without that bit being set.

RETURN VALUE

0 The dg_mknod operation was successful.

-1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EEXIST	The named file *path* already exists.
EINVAL	*version* is not a supported version of struct dg_mknod.
EINVAL	An invalid file type was specified in the *buffer_ptr* extended_mode.
EROFS	The directory in which *path* is to be created is located on a file system device that is mounted read-only.

ENOSPC	There is not enough contiguous space available to allocate file space or an inode.
EFAULT	*buffer_ptr* points to an invalid address, or the pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname of the target of the symbolic link being created exceeds the length limit for pathnames, or A component of the pathname of the target of the symbolic link being created exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	Permission to create a character-special file or a block-special file is denied, or the pathname contains a character not in the allowed character set.
EACCES	The calling process does not have permission to resolve the pathname.

SEE ALSO

`chmod`(2), `chown`(2), `creat`(2), `dg_fstat`(2), `dg_mstat`(2), `fchmod`(2), `fchown`(2), `fstat`(2), `link`(2), `lstat`(2), `mknod`(2), `pipe`(2), `read`(2), `stat`(2), `time`(2), `unlink`(2), `utime`(2), `utimes`(2), `write`(2), `dg_mknod`(5), `dg_stat`(5), `stat`(5).

NAME

dg_mount – mount a file system

SYNOPSIS

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

int  dg_mount (type, path, flags, data)
char * type;
char * path;
int  flags;
char * data;
```

where:

type	Address of a type string (must be nfs or dg/ux)
path	Address of a pathname of a file to mount upon
flags	Mount options flags
data	Type-specific argument structure

DESCRIPTION

The dg_mount system call is used to mount all file system types. The dg_mount call attaches a file system to a file. When mounting a dg/ux or nfs file system, *path* must refer to a directory or CPD. After a successful return, references to *path* will refer to the root directory on the newly mounted file system. When mounting a namefs file system, *path* may be any type of file. After a successful return, references to *path* will refer to the named stream.

The following option flags are supported when mounting dg/ux and nfs file systems:

M_RDONLY Mount the file system read-only.

M_NOSUID Ignore set-uid bits on files in this file system.

M_REMOUNT Change the options on an existing mount. For NFS file systems, the following mount options may have their values changed by this flag: *wsize*, *rsize*, *timeo*, *retrans*, *acregmin*, *acregmax*, *acdirmin*, *acdirmax*, honoring of set-uid bits on files on this filesystem, and how the file system is mounted (that is, hard or soft).

M_NOSUB Disallow mounts beneath this filesystem.

Physically write-protected file systems must be mounted read-only; otherwise, errors will occur when access times are updated, whether or not any explicit write is attempted.

These flags are ignored when mounting a namefs file system.

The *type* string indicates the type of the filesystem. *data* is a pointer to a structure that contains the type-specific arguments to dg_mount. Below is a list of the filesystem types supported and the type-specific arguments to each:

```
"dg/ux"
struct dgux_args {
        int                 version;
        char                *fspec;
        int                 flags;
        int                 file_nodes;
```

```
        int                     file_space;
        mode_t                  permissions;
        int                     log_size;
        char                    *cachespec;
};

"nfs"
struct nfs_args {
        int                     version;
        struct sockaddr_in      *addr;
        fhandle_t               *fh;
        int                     flags;
        int                     wsize;
        int                     rsize;
        int                     timeo;
        int                     retrans;
        char                    *hostname;
        int                     acregmin;
        int                     acregmax;
        int                     acdirmin;
        int                     acdirmax;
        char                    *netname;
        int                     securewin;
};

"namefs"
struct namefs_args {
        int                     fd;
};
```

For dg/ux file systems, the *version* must be DG_MOUNT_DGUX_VERSION, and *fspec* points to a character string that names the block special device being mounted. If the variant of the dg/ux mount is for a memory file system, three additional flags come into play. A memory file system is one that has no underlying media. Files created in a memory file system will not persist across system instantiations. Memory file systems are useful for storing temporary files and for accelerating executable images. *Permissions* is the mode to assign to file systems that emulate DG/UX file systems on top of other file systems (not currently used).

The additional flags for the memory file system variant are:

DGUXMNT_MEMORY_FS
: The mount is for a file system that does not have any backing media, that is, one whose file information and data exist in the virtual memory of the system. If this is set, the next two flags may also be defined. If this flag is not set, the following two flags are ignored.

DGUXMNT_WIRED_MEMORY
: This instructs VM to use wired memory for the data in the memory file system instead of unwired memory, which is the default.

DGUXMNT_FILE_COUNT
: The *file_nodes* member of the structure contains the maximum number of files allowed to be allocated to the particular memory file system. *file_nodes* must be a positive integer. If this is not specified, the default file count for

the memory file system is 16384.

DGUXMNT_FILE_SPACE
: The *file_space* member of the structure contains the maximum amount of file space allowed to be allocated in the particular memory file system. *file_space* must be a positive integer. If this is not specified, the default amount of file space for the memory file system is 2048 blocks, where a block is 512 bytes.

For NFS file systems, the *version* must also be DG_MOUNT_NFS_VERSION. The *addr* socket contains the UDP address of the NFS file server. The *fh* file handle contains the file handle on the server of the root of the file system being mounted. The *flags* word is the logical OR of any of these flags:

NFSMNT_SOFT	The requested mount should be a soft mount.
NFSMNT_WSIZE	The *wsize* member of the structure contains the maximum transfer size (in bytes) to use when writing files. If no value is specified, 8192 is used.
NFSMNT_RSIZE	The *rsize* member of the structure contains the maximum transfer size (in bytes) to use when reading files. If no value is specified, 8192 is used.
NFSMNT_RETRANS	The *retrans* member of the structure contains a retransmission count for NFS retrys. If no value is specified, 3 is used.
NFSMNT_NOAC	Disable attribute caching for all files and directories.
NFSMNT_ACREGMIN	The *acregmin* member of the structure contains a minimum number of seconds to keep attributes cached for regular files. If no value is specified, 3 seconds is used.
NFSMNT_ACREGMAX	The *acregmax* member of the structure contains a maximum number of seconds to keep attributes cached for regular files. If no value is specified, 60 seconds is used.
NFSMNT_ACDIRMIN	The *acdirmin* member of the structure contains a minimum number of seconds to keep attributes cached for directory files. If no value is specified, 30 seconds is used.
NFSMNT_ACDIRMAX	The *acdirmax* member of the structure contains a maximum number of seconds to keep attributes cached for directory files. If no value is specified, 60 seconds is used.

For namefs file systems, *fd* is an open file descriptor that refers to a STREAMS-based pipe or a STREAMS device driver. The mount attaches the stream to *path* so that all subsequent operations on *path* will operate on the named stream. The *flags* word is ignored.

ACCESS CONTROL

The effective user id of the calling process must be superuser to mount a dg/ux or nfs file system. When mounting a namefs file system, the effective user id of the calling process must be superuser, or the the effective user id must be the owner of *path* and have write access to *path*.

RETURN VALUE

0 Completed successfully.

−1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

ENOTDIR	*path* is not a directory.
EPERM	Permission to mount a file system device is denied to the calling process.
EBUSY	Another process is using *path* as its home or root directory.
EBUSY	Another file system is already mounted here.
EINVAL	The version number in the filesystem specific packet is not correct.
ENODEV	Kernel support for the requested file system type is not present.

Any of the pathname resolution errors.

SEE ALSO

`mount(1M)`, `getfh(2)`, `mount(2)`, `umount(2)`.

NAME

`dg_mstat` – get file status

SYNOPSIS

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

int    dg_mstat (path, buffer_ptr)
char   * path;
struct stat * buffer_ptr;
```

where:

path	Address of a pathname
buffer_ptr	Address of a `stat` buffer to fill

DESCRIPTION

`Dg_mstat` returns the current attributes of the file pointed to by *path* into the status buffer at the location specified by *buffer_ptr*. If *path* refers to a symbolic link, file status for the target of the symbolic link is returned. Furthermore, if *path* (after symbolic link resolution, if any) refers to a mount point for a file system, status information for the mounted on directory is returned.

The interpretation of the file's attributes depends on the file's type [see `stat(5)` for details]. The subject file must be of type 'ordinary-disk-file', 'directory', 'block-special-file', 'character-special-file', or 'fifo-special-file'.

If `dg_mstat` fails, the contents of the buffer are undefined.

ACCESS CONTROL

Read, write, or execute permission of the named file is not required, but the process must have permission to resolve *path*.

RETURN VALUE

0	The `dg_mstat` operation was successful.
–1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EFAULT	*Buffer_ptr* points to an invalid address.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.

EFAULT The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

chmod(2), chown(2), creat(2), fchmod(2), fchown(2), fstat(2), link(2), lstat(2), mknod(2), pipe(2), read(2), stat(2) time(2), unlink(2), utime(2), utimes(2), write(2), stat(5).

NAME

dg_paging_info – determine residency of memory pages

SYNOPSIS

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

int dg_paging_info(int version, pid_t pid,
                   struct dg_paging_info *paging_info);
```

where:

version Desired version of the function interface

pid Process id of the process whose address space is to be queried

paging_info Pointer to a structure which further describes the queried memory region and the array used for reporting paging status

DESCRIPTION

The dg_paging_info() function returns the primary memory residency status for pages in a region of the address space specified by *pid*.

The *version* parameter must have the value DG_PAGING_INFO_VERSION_1, as this is the only supported version of the function interface.

The *pid* parameter must be either a legal process id value, to query the address space of the process with that process id, or one of two special values:

DG_PAGING_INFO_KERNEL_SPACE_PID
: Query a region of the kernel address space.

DG_PAGING_INFO_CALLING_PROCESS_PID
: Query a region of the caller's address space.

Upon invocation of dg_paging_info(), the members of the structure given by *paging_info* further define the query:

dpi_flags This member is reserved for future use and must be 0.

dpi_start_address This member defines the lower bound within the queried address space for which the caller is requesting paging info; its value must be a page aligned address. (The system page size is available by calling getpagesize(2) or sysconf(2) with the _SC_PAGESIZE parameter; both calls return identical values.)

dpi_byte_count This member contains the maximum number of bytes of address space for which the caller is requesting paging info. In the event that the value is not a multiple of the system page size, it will be treated as if it were rounded up to the next page size multiple.

dpi_bitmap_bits_per_page This member contains the number of bits of paging info requested by the caller for each page of the queried address space. The legal values for this member are 1 and 8. In either case, the low order bit of paging info recorded for each page indicates whether it is resident in primary memory (a bit value of 1 represents a resident page). In the case of one bit per page, the pages with lower addresses are represented in the higher order bits within each byte.

dpi_bitmap_ptr This member specifies the location within the caller's address space of the bitmap to contain the recorded paging info for the queried address space. The caller's address space must be writable starting at the specified address, for the number of bits of paging info requested. (This number of bits can be computed by multiplying the number of pages implied by *dpi_byte_count* times the value of *dpi_bitmap_bits_per_page*.)

The function will report on the first contiguous range of mapped pages at or above address *dpi_start_address* within the queried address space, up to the maximum number of pages implied by *dpi_byte_count*.

Upon successful return from `dg_paging_info`, the following members of the *paging_info* structure will be updated to reflect what status information has been reported:

dpi_start_address This member contains the address of the first mapped page within the queried address space which has an address greater than or equal to the requested start address. This page's status is also represented by the first element in the bitmap at *dpi_bitmap_ptr*.

dpi_byte_count This member contains the actual number of bytes of address space for which paging info has been reported in the bitmap. This value will be identical to its value upon invocation unless the memory segment starting at *dpi_start_address* (as returned) is smaller than the value of *dpi_byte_count* specified upon invocation. If *dpi_byte_count* is `0` on return, then there were no mapped pages in the queried address space at or above *dpi_start_address*.

The `dg_paging_info()` function returns residency information that is accurate at a different instant in time for each page. Because the system may frequently adjust the set of pages in memory, this information may quickly be outdated, not necessarily even self-consistent.

Pages which are direct mapped to a memory-mapped device will be reported by `dg_paging_info()` to be memory resident.

ACCESS CONTROL

If the queried address space is that of a process, then the caller's real user id or saved set user id must equal the real user id or saved set user id of the queried process. Failing that, the caller's effective user id must be superuser.

If the queried address space is the kernel address space, then the caller's effective user id must be superuser.

RETURN VALUE

Upon successful completion, `dg_paging_info()` returns a value of `0`. Otherwise, it returns the value `-1`, and sets `errno` to indicate the error.

DIAGNOSTICS

Under the following conditions, `dg_paging_info()` fails and sets `errno` to:

EACCES — if the calling process lacks access to query the address space specified by *pid*.

EFAULT — if some portion of the structure pointed to by *paging_info* is not mapped in the caller's address space or lacks write access.

EFAULT — if some portion of the bitmap pointed to by *dpi_bitmap_ptr* is not mapped in the caller's address space or lacks write access.

EINVAL — if *version* is not DG_PAGING_INFO_VERSION_1.

EINVAL — if *dpi_flags* is not 0.

EINVAL — if *dpi_start_address* is not a page aligned address.

EINVAL — if *dpi_byte_count* is 0.

EINVAL — if *dpi_bitmap_bits_per_page* is neither 1 nor 8.

ESRCH — if no process was found with a process id matching *pid*.

SEE ALSO

getpagesize(2), mincore(2), sysconf(2).

NAME

dg_process_info – get information about the system's currently active processes

SYNOPSIS

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

int     dg_process_info (selector,
        selector_value,
        cmd_name_format,
        key_ptr,
        buffer_ptr,
        version)
long    selector;
long    selector_value;
long    cmd_name_format;
long    * key_ptr;
struct dg_process_info * buffer_ptr;
long    version;
```

where:

selector	A condition for determining which process to report information about
selector_value	A value for the select condition above
cmd_name_format	One of three values (DG_PROCESS_INFO_CMD_NAME_NULL, DG_PROCESS_INFO_CMD_NAME_ONLY, or DG_PROCESS_INFO_CMD_NAME_AND_ARGS) specifying whether the command name alone is sufficient, or if the command name with its arguments is needed.
key_ptr	On the first call, a pointer to a variable containing the value DG_PROCESS_INFO_INITIAL_KEY; on return, a handle that should be used on subsequent calls to this system call
buffer_ptr	A pointer to a buffer of structure dg_process_info where the process information will be returned
version	To use the most recent version, *version* should be set to DG_PROCESS_INFO_CURRENT_VERSION.

DESCRIPTION

The dg_process_info system call searches the process table for valid (non-free non-initializing) table entries and based on the *selector* determines whether or not to return information about that process. Searching continues until a process is found (return value (1)), or until the process table is completely searched (return value (0)). Only one search through the process table is made. If a process exists in a particular slot in the process table when that slot is looked at, information on that process will be returned. If the slot is empty, no information will be returned.

ACCESS CONTROL

There are no checks for access control. All users can access this system call.

RETURN VALUE

1 Successful search, but not done. The *buffer* was filled with information about a process. *key_ptr* is given a handle which subsequent calls to this routine will use to continue the search through the process table.

0 Successful completion, that is, no more processes. The contents of *buffer* are undefined.

-1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EINTR	An interrupt occurred during the system call
EINVAL	A bad argument was passed in.
EFAULT	The *key_ptr* argument specifies a bad address
EFAULT	The *buffer_ptr* argument specifies a bad address

SEE ALSO

killall(1M), dg_sys_info(2), fork(2), vfork(2).

NAME

dg_set_cpd_limits – change the resource limits of a control point directory

SYNOPSIS

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

int dg_set_cpd_limits (path, blocks, file_nodes)
char * path;
unsigned long blocks;
unsigned long file_nodes;
```

where:

path	Pathname of the CPD to be changed
blocks	New block allocation ceiling
file_nodes	New file node allocation ceiling

DESCRIPTION

The dg_set_cpd_limits system call changes the limits associated with a control point directory. The *path* parameter points to a pathname naming a control point directory (terminal symbolic links are followed in *path*). If the calling process has the appropriate access to *path* (see below), the CPD limits of *path* are set to *blocks* disk blocks and *file_nodes* file nodes. The new CPD limits will be visible in subsequent dg_stat calls on *path:* the *max_cpd_blocks* field will be *blocks* and the *max_cpd_file_nodes* field will be *file_nodes*.

The effects of a CPD's limits on its space descendants (those files and directories which are below *path* and not across a file system mount point boundary from it) are as follows: If the current number of disk blocks used by *path* and all its space descendants equals or exceeds *blocks*, all attempts to allocate blocks to *path* or one of its space descendants will fail with the error ENOSPC. Likewise, if the total number of file nodes used by *path* and all its space descendants equals or exceeds *file_nodes*, all attempts to allocate more file nodes below *path* will fail with the error ENOSPC. The only exception to these rules is that the superuser may override the CPD limits of the root directory of a file system (which is always a CPD), though obviously not in excess of the actual physical resources in the file system.

The *blocks* parameter can be set to any number between 0 and DG_CPD_NO_BLOCK_LIMIT, inclusive. Likewise, the *file_nodes* parameter can be set to any number between 0 and DG_CPD_NO_FILE_NODE_LIMIT, inclusive. Note that it is not required that *blocks* be greater than the current number of blocks in use by *path* and its space descendants, or that *file_nodes* be greater than the current number of file nodes in use by *path* and its space descendants.

The last component of *path* may not be "." or "..". Use an absolute pathname instead.

ACCESS CONTROL

The calling process must have write access to the parent directory of *path*, unless *path* is the root of a file system. In that case, only the superuser may make this call.

The process must have permission to resolve *path* .

RETURN VALUE

0	The limits of the control-point directory *path* were successfully modified.
−1	An error occurred. errno is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EINVAL	The named file *path* exists but it is not a control point directory; or the *blocks* parameter is not in the range 0 to DG_CPD_NO_BLOCK_LIMIT; or the *file_nodes* parameter is not in the range 0 to DG_CPD_NO_FILE_NODE_LIMIT; or the last component of the *path* is "." or "..".
ENOTSUPPORTED	The operation is not supported because the referenced file is an NFS file.
EPERM	The CPD denoted by *path* is the root of a file system, and the process's effective user-ID is not superuser.
EACCES	The CPD denoted by *path* is not the root of a file system, and the process does not have write permission in the parent directory of *path*.
EROFS	The named file resides on a file system device mounted read-only.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.
EACCES	The calling process does not have permission to resolve the pathname.

SEE ALSO

`mkdir`(1), `rmdir`(1), `dg_mknod`(2), `rmdir`(2), `dg_stat`(5).

NAME

dg_setsecretkey – store a client's secret key in the keyserver

SYNOPSIS

```
int dg_setsecretkey (secretkey)
char * secretkey;
```

where:

secretkey The secret key

DESCRIPTION

This call is used to store a user's decrypted secret key in the database maintained by the keyserver process.

ACCESS CONTROL

None.

RETURN VALUE

0 The operation was successful.

–1 An error occurred. errno indicates the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EINVAL Secure RPC is not configured. Secure RPC using DES Authentication is an additional feature that must be purchased separately from the DG/UX™ ONC™/NFS® product.

EFAULT Some part of the string pointed to by *secretkey* lies outside the process's readable address space.

SEE ALSO

dg_decryptsessionkey(2), dg_encryptsessionkey(2), dg_getrootkey(2).

NAME

dg_stat – get extended file status information

SYNOPSIS

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

int  dg_stat (path, buffer_ptr, link_intent, version)
char * path;
struct dg_stat * buffer_ptr;
int     link_intent;
unsigned short version;
```

where:

path	Address of a pathname
buffer_ptr	Address of a dg_stat buffer to fill
link_intent	Instructions on what to do if the component of *path* is a symbolic link
version	Version of the struct dg_stat packet that *buffer_ptr* refers to; should be set to DG_STAT_VERSION_NUMBER

DESCRIPTION

Dg_stat(2) returns the current extended attributes of the file named by *path* into the dg_stat buffer at the location specified by *buffer_ptr*. If *path* refers to a symbolic link and *link_intent* is DG_STAT_FOLLOW_SYMLINK, then file status for the target of the symbolic link is returned. If *path* refers to a symbolic link and *link_intent* is DG_STAT_EXAMINE_SYMLINK, then file status for the symbolic link itself is returned. If *path* does not refer to a symbolic link, then the value of *link_intent* is irrelevant, but it must be one of the aforementioned constants. If dg_stat fails, the contents of the buffer are undefined.

The size and composition of the structure referred to by *buffer_ptr* is determined by the *version* parameter. All calls to this function should use DG_STAT_VERSION_NUMBER for this parameter. *version* allows for future revisions of struct dg_stat to be handled in a compatible way.

The interpretation of the file's attributes depends on the file's type [see dg_stat(5) and stat(5)].

ACCESS CONTROL

Read, write, or execute permission of the named file is not required, but the process must have permission to resolve *path*.

RETURN VALUE

0 The dg_stat operation was successful.

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EINVAL	*version* is not a supported version of struct dg_stat.
EINVAL	*link_intent* is not one of DG_STAT_EXAMINE_SYMLINK or DG_STAT_FOLLOW_SYMLINK.
EFAULT	*buffer_ptr* points to an invalid address.

ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.
EACCES	The calling process does not have permission to resolve the pathname.

SEE ALSO

chmod(2), chown(2), creat(2), dg_fstat(2), dg_mstat(2), fchmod(2), fchown(2), fstat(2), link(2), lstat(2), mknod(2), pipe(2), read(2), stat(2), time(2), unlink(2), utime(2), utimes(2), write(2), dg_stat(5), stat(5).

NAME

dg_sys_info – get system information

SYNOPSIS

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

int   dg_sys_info (info_ptr,
      info_type,
      version)
long  * info_ptr;
long  info_type;
long  version;
```

where:

info_ptr	A pointer to the location in user space where the information will be written
info_type	The type of structure pointed to by *info_ptr*. See sys/dg_sys_info.h.
version	The version of the particular sys_info structure being used

DESCRIPTION

Based on the *info_type* of *info_ptr*, gather information from certain kernel databases and return the information to the caller. See sys/dg_sys_info.h for explanations fo the types of information returned.

RETURN VALUE

0	The call succeeded.
–1	An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EINVAL	A bad argument was passed.
EFAULT	The *info_ptr* argument specifies a bad address.
EONOTSUPP	Kernel support for NFS is not present.
ENOMEM	Not enough memory for internal kernel structure.

SEE ALSO

dg_process_info(2), uname(2).

NAME

dg_sysctl – perform system configuration and control functions

SYNOPSIS

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

int          dg_sysctl (cmd, arg)
unsigned int cmd;
void         * arg;
```

where:

cmd The task to be performed.

arg A pointer to a packet of information used by and/or filled in by the task.

DESCRIPTION

The dg_sysctl system call can be used to perform a variety of system configuration and system control tasks. The specific task to be executed is indicated by the *cmd* parameter, and the address of an information packet used by and/or filled in by that command is passed in the *arg* parameter. The various command values and the types of their accompanying argument packets are defined and described in <sys/dg_sysctl.h>.

Commands

The DG_SYSCTL_CONFIGURE_DEVICE command is used to configure a device into the system, given only its name in DG/UX common device specification format.

The DG_SYSCTL_DECONFIGURE_DEVICE command is used to deconfigure a device out of the system, given only its name in DG/UX common device specification format.

The DG_SYSCTL_NAME_TO_DEVICE command is used to find out the device number of a device, given only its name in DG/UX common device specification format.

The DG_SYSCTL_DEVICE_TO_NAME command is used to find out the canonical DG/UX common device specification format name of a device, given only its device number.

The DG_SYSCTL_SET_BOOT_PATH command is used to set the boot command line that will be used to reboot the system if the reboot(2) or uadmin(2) system call is invoked appropriately, or if the system is automatically rebooted after a panic. The default is the boot path used when the system was last booted. If the boot path is set to an empty string or a string of spaces, the boot path saved by the System Control Monitor (SCM) will be used. The boot path will be used for all future automatic reboots of the system until you change the boot path or reboot the system manually from the SCM. Do not include the SCM boot command in the string itself.

The DG_SYSCTL_GET_BOOT_PATH command is used to get the current boot command line so that you can see what it is.

The DG_SYSCTL_SET_DUMP_DEVICE command is used to set the name of the dump device that will be used to write a system dump when a panic occurs. The default dump device is the device specified for the DUMP variable in the system configuration file.

The DG_SYSCTL_GET_DUMP_DEVICE command is used to get the the current dump device name so that you can see what it is.

The DG_SYSCTL_SET_AUTOREBOOT command is used to set the state that controls whether the system will be automatically rebooted after a panic occurs. The default state is DG_SYSCTL_HALT_AFTER_PANIC. The other state that it can be set to is DG_SYSCTL_REBOOT_AFTER_PANIC.

The DG_SYSCTL_GET_AUTOREBOOT command is used to get the current state setting that controls whether the system will be automatically rebooted after a panic occurs so that you can see what it is.

The DG_SYSCTL_SET_DUMP_START command is used to set the state that controls how a system dump is to be started when a panic occurs. The default state is DG_SYSCTL_ASK_FOR_DUMP. The other states that it can be set to are DG_SYSCTL_AUTO_DUMP and DG_SYSCTL_SKIP_DUMP.

The DG_SYSCTL_GET_DUMP_START command is used to get the current state setting that controls how a system dump is to be started when a panic occurs so that you can see what it is.

Automatic Panic Dumps

If you set the dump start state to DG_SYSCTL_AUTO_DUMP, you need to be aware of these things:

- You must ensure that the dump medium is always available and ready to be used. For tape drives, this means that you must have a write-enabled tape in the drive at all times.

  If you need to use the dump medium for some other purpose, you should first disable automatic panic dumps or change the dump device to some other available device. After you are finished using the device, you can re-enable automatic panic dumps or switch the dump device back. If you do not make either of these temporary changes and a panic occurs, the medium in the dump device will be overwritten.

- If a panic occurs and the dump device cannot be opened (e.g., no tape in the drive), the dump will be skipped instead.

- As long as the dump starts and completes successfully with the available medium, no operator intervention is required.

  However, if there are any further problems with the dump (e.g., hard error on the tape, new tape volume required for a multi-volume dump), the operator will be prompted to mount a new tape and respond once the tape is ready.

Operator Shutdowns

If you use the hot key sequence to cause a system panic or use the "s 1000" SCM command to initiate an operator shutdown, the autoreboot and dump start states will be reset to their default values of DG_SYSCTL_HALT_AFTER_PANIC and DG_SYSCTL_ASK_FOR_DUMP, respectively. This will give the operator complete control over the system during the panic processing.

Skipping Panic Dumps

If you set the dump start state to DG_SYSCTL_SKIP_DUMP and the autoreboot state is set to DG_SYSCTL_HALT_AFTER_PANIC and a panic occurs, the panic dump will be skipped and the system will be halted just as expected. After the system has halted, if you change your mind and decide that you do want to take a system dump, type "s 1000" at the SCM prompt and you will then be asked whether you want to take a system dump.

String Lengths

The maximum string length for a device name returned by the DG_SYSCTL_DEVICE_TO_NAME and DG_SYSCTL_GET_DUMP_DEVICE commands is 256 characters, including the trailing null.

The maximum string length for a boot path returned by the DG_SYSCTL_GET_BOOT_PATH command is 256 characters, including the trailing null.

EXAMPLES

For the DG_SYSCTL_CONFIGURE_DEVICE, DG_SYSCTL_DECONFIGURE_DEVICE, DG_SYSCTL_SET_BOOT_PATH, and DG_SYSCTL_SET_DUMP_DEVICE commands, you specify the address of a string as the *arg* parameter. Here are some examples of using these commands:

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

int status;

status = dg_sysctl (DG_SYSCTL_CONFIGURE_DEVICE,
                    "duart(1)");

status = dg_sysctl (DG_SYSCTL_DECONFIGURE_DEVICE,
                    "inen()");

status = dg_sysctl (DG_SYSCTL_SET_BOOT_PATH,
                    "sd(cisc(),0)/dgux -3");

status = dg_sysctl (DG_SYSCTL_SET_DUMP_DEVICE,
                    "st(insc(),4)");
```

For the DG_SYSCTL_NAME_TO_DEVICE command, allocate and fill in a dg_sysctl_name_to_device packet and pass its address as the *arg* parameter. The device number will be returned in the device_number field of the packet. Here is an example of using this command:

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

int    status;
struct dg_sysctl_name_to_device name_to_device_pkt;

name_to_device_pkt.device_name = "sd(insc(),0)";

status = dg_sysctl (DG_SYSCTL_NAME_TO_DEVICE,
                    &name_to_device_pkt);

printf ("device number is %d\n",
        name_to_device_pkt.device_number);
```

For the DG_SYSCTL_DEVICE_TO_NAME command, allocate and fill in a dg_sysctl_device_to_name packet and pass its address as the *arg* parameter. The device name will be returned in a string you allocate. Pass the address of that string in the device_name field of the packet. Here is an example of using this command:

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

int    status;
char   returned_device_name [256];
```



```
struct dg_sysctl_device_to_name device_to_name_pkt;

device_to_name_pkt.device_number   = 393216;
device_to_name_pkt.device_name     = returned_device_name;
device_to_name_pkt.max_name_length = 256;

status = dg_sysctl (DG_SYSCTL_DEVICE_TO_NAME,
                    &device_to_name_pkt);

printf ("device name is '%s'\n",
        returned_device_name);
```

For the DG_SYSCTL_GET_BOOT_PATH command, allocate and fill in a dg_sysctl_get_boot_path packet and pass its address as the *arg* parameter. The boot path will be returned in a string you allocate. Pass the address of that string in the boot_path field of the packet. Here is an example of using this command:

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

int    status;
char   returned_boot_path [256];
struct dg_sysctl_get_boot_path get_boot_path_pkt;

get_boot_path_pkt.boot_path       = returned_boot_path;
get_boot_path_pkt.max_name_length = 256;

status = dg_sysctl (DG_SYSCTL_GET_BOOT_PATH,
                    &get_boot_path_pkt);

printf ("boot path is '%s'\n",
        returned_boot_path);
```

For the DG_SYSCTL_GET_DUMP_DEVICE command, allocate and fill in a dg_sysctl_get_dump_device packet and pass its address as the *arg* parameter. The dump device will be returned in a string you allocate. Pass the address of that string in the dump_device_name field of the packet. Here is an example of using this command:

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

int    status;
char   returned_dump_device [256];
struct dg_sysctl_get_dump_device get_dump_device_pkt;

get_dump_device_pkt.dump_device_name = returned_dump_device;
get_dump_device_pkt.max_name_length  = 256;

status = dg_sysctl (DG_SYSCTL_GET_DUMP_DEVICE,
                    &get_dump_device_pkt);

printf ("dump device name is '%s'\n",
        returned_dump_device);
```


For the DG_SYSCTL_SET_AUTOREBOOT, DG_SYSCTL_GET_AUTOREBOOT, DG_SYSCTL_SET_DUMP_START, and DG_SYSCTL_GET_DUMP_START commands, you specify the address of an unsigned int as the *arg* parameter. For the two "set" commands, fill in the unsigned int with the value you want to set the system option to before calling dg_sysctl. For the two "get" commands, dg_sysctl will return the current setting of the system option in the unsigned int. Here are some examples of using these commands:

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

int          status;
unsigned int option_value;

option_value = DG_SYSCTL_REBOOT_AFTER_PANIC;

status = dg_sysctl (DG_SYSCTL_SET_AUTOREBOOT,
                    &option_value);

option_value = DG_SYSCTL_AUTO_DUMP;

status = dg_sysctl (DG_SYSCTL_SET_DUMP_START,
                    &option_value);

status = dg_sysctl (DG_SYSCTL_GET_AUTOREBOOT,
                    &option_value);

printf ("autoreboot option value is %u\n",
        option_value);

status = dg_sysctl (DG_SYSCTL_GET_DUMP_START,
                    &option_value);

printf ("dump start option value is %u\n",
        option_value);
```

For more details on the dg_sysctl system call input and output arguments for each command, see the <sys/dg_sysctl.h> include file.

ACCESS CONTROL

Any user may execute the DG_SYSCTL_NAME_TO_DEVICE, DG_SYSCTL_DEVICE_TO_NAME, DG_SYSCTL_GET_BOOT_PATH, DG_SYSCTL_GET_DUMP_DEVICE, DG_SYSCTL_GET_AUTOREBOOT, and DG_SYSCTL_GET_DUMP_START commands.

Only the superuser may execute the DG_SYSCTL_CONFIGURE_DEVICE, DG_SYSCTL_DECONFIGURE_DEVICE, DG_SYSCTL_SET_BOOT_PATH, DG_SYSCTL_SET_DUMP_DEVICE, DG_SYSCTL_SET_AUTOREBOOT, and DG_SYSCTL_SET_DUMP_START commands.

RETURN VALUE

0 The dg_sysctl operation was successful.

-1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EPERM	A process called dg_sysctl and attempted to execute a restricted command without being the superuser.
ENXIO	An attempt was made to configure a device that was already configured.
ENXIO	An attempt was made to deconfigure, get the name of, or get the device number of a device that is not configured.
ENXIO	An attempt to configure or deconfigure a device failed for an unknown reason.
ENOMEM	There was insufficient kernel memory available to execute *cmd*.
EFAULT	*arg* or a pointer in the packet points to an invalid address.
EFAULT	An attempt was made to configure or deconfigure a device, get the device number of a device from its name, set the boot command line, or to set the dump device name, but the string specified was too long.
EBUSY	An attempt was made to deconfigure a busy or undeconfigurable device.
EINVAL	*cmd* is not one of the valid commands described above.
EINVAL	An attempt was made to configure or deconfigure a device, get the device number of device from its name, or to set the dump device name, but the device name did not conform to the DG/UX Common Device Specification Format.
EINVAL	An attempt was made to get the name of a device from its device number, the boot command line, or the dump device name, but not enough string storage was allocated to receive the name.
EINVAL	An attempt was made to set the auto-reboot or dump start state, but the value pointed to by *arg* was not one of the valid values for the commands which are specified in <sys/dg_sysctl.h>.

SEE ALSO

dg_sysctl(1M), diskman(1M), reboot(1M), reboot(2), uadmin(2).

NAME

`dg_unbuffered_read` – synchronously read data from a file without system buffering

SYNOPSIS

```
int dg_unbuffered_read (fildes, buffer, start_block, num_blocks)
int       fildes;
char      buffer[];
unsigned  start_block;
unsigned  num_blocks;
```

where:

fildes	A valid descriptor
buffer	User data buffer
start_block	Starting logical block number
num_blocks	Size (in blocks) of the read

DESCRIPTION

The `dg_unbuffered_read` system call transfers *num_blocks* blocks of data from the file associated with *fildes* into the buffer pointed to by *buffer*. The starting block number of the transfer is given by the *start_block* parameter.

The file position attribute of *fildes* is ignored by this interface. The starting block position of the read must be specified in the call. The file position attribute of *fildes* remains unchanged by the execution of this call.

If mandatory record locking is enabled for the file, this call will fail.

ACCESS CONTROL

The *fildes* must have been opened for unbuffered I/O with the O_DG_UNBUFFERED flag, and *fildes* must also have been opened for read access.

RETURN VALUE

0..*num_blocks*
: Completed successfully. The number of blocks actually read is returned. The value 0 indicates EOF (end-of-file).

–1
: An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	*fildes* is not opened for unbuffered reading.
EFAULT	*buffer* points outside the allocated address space.
EINTR	A signal was caught during the system call.

SEE ALSO

`open(2)`, `dg_unbuffered_write(2)`.

NAME

`dg_unbuffered_write` – synchronously write data to a file without system buffering

SYNOPSIS

```
int dg_unbuffered_write(fildes, buffer, start_block, num_blocks)
int      fildes;
char     buffer[];
unsigned start_block;
unsigned num_blocks;
```

where:

fildes	An active descriptor
buffer	User data buffer
start_block	Starting logical block of request
num_blocks	Size (in blocks) of the request

DESCRIPTION

The `dg_unbuffered_write` system call transfers *num_blocks* blocks of data from the buffer pointed to by *buffer* into the object associated with *fildes*. The starting position of the request is given by *start_block*.

The file position attribute of *fildes* is ignored by this interface. The starting block position of the write must be specified in the call. The file position attribute of *fildes* remains unchanged by the execution of this call.

If the operation is successful, the inode is flushed out before control is returned to the user if any attribute other than *time_last_accessed*, *time_last_modified*, or *time_last_changed* of the file has changed. Index blocks modified by this operation are also flushed before returning control to the user.

If mandatory record locking is enabled for the file, this call will fail.

ACCESS CONTROL

The process must have write access to the descriptor *fildes*, and it must have been opened with the O_DG_UNBUFFERED open flag to allow for unbuffered I/O access.

RETURN VALUE

0..*num_blocks*
Completed successfully. The number of blocks actually written is returned.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	*fildes* is not opened for unbuffered writing.
EFBIG	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.
EFAULT	*Buffer* points outside the allocated address space.
EINTR	A signal was caught during the system call.

SEE ALSO

`dg_unbuffered_read`(2), `open`(2), `write`(2).

NAME

`dg_xtrace` – extended process trace

SYNOPSIS

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

int   dg_xtrace (request, data)
int   request;
union dg_xtrace_u * data;
```

where:

request Process trace command of the form `DG_XT_`*name*, where *name* is a unique suffix indicating the action to be taken

data Pointer to union used for completing the process trace command

DESCRIPTION

`Xtrace` lets a process (debugger process) control the execution of another process (target process). Its primary use is to implement breakpoint debugging; see `sdb`(1) and `dbx`(1). It is an extended version of `ptrace` that has been added to remedy two of the major shortcomings of `ptrace`:

- the mandatory parent/child relationship between the tracing-process and the process-being-traced, and
- the very small (32-bit) interface between the two processes.

Since the relationship between the two processes is no longer mandated to be parent/child some terms defining this new relationship are needed so that discussion of their interaction is possible. The process controlling the tracing of another process will be referred to as the "controlling process" or "controller." The process being traced will be referred to as the "target process" or "target."

Tracing Relationships

Three valid tracing relationships exist between the controlling and target processes:

- the controller is the PARENT of the target,
- the controller is the CHILD of the target, or
- the controller is NOT RELATED to the target (and is not the target itself).

Some restrictions exist concerning the specifics of the tracing relationship. In general, a target can become a controller, but a controller can not become a target. This avoids a cycle in the controller-target relationship which could result in a deadlock condition. A controller can trace multiple targets, but when the controller terminates, all of its targets are sent SIGKILL. Finally, if a controller (who is not a target) is the child of a process being traced then job control stop signals sent to the controller will be ignored. This restriction is necessary because the parent process is usually responsible for any stopped children (which is impossible in this scenario).

In all configurations, the target process behaves normally until it encounters a signal (see `signal`(2) for the list) or until it exits; it then stops for tracing and its controller is notified via `dg_xtrace` using the request XT_WAIT_FOR_TARGET (defined below). (A signal that is blocked does not cause the process to stop for tracing until the signal is unblocked. If a job control stop signal is held because the target process has performed `vfork`(2) but not `exec`(2) or `exit`(3C), such a signal does not cause the process to stop for tracing until the process has performed `exec` or `exit`.)

When the target is stopped, its controller can examine and modify its address space using `dg_xtrace`. Also, the controller can cause the target either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

In the first tracing relationship, the sequence of events required to trace a process is as follows:

1. The target process is created by `fork`(2) or `vfork`(2).
2. The target process performs a `dg_xtrace` operation with *request* DG_XT_TRACE_ME.
3. The target's address space is changed by the `exec`(2) operation. This causes the target to be stopped for tracing before executing the first instruction of the new image as if the signal SIGTRAP had occurred.
4. The controlling process waits for the target to stop for tracing using `dg_xtrace` with *request* DG_XT_WAIT_FOR_TARGET.
5. The controller may now cause the target to continue execution using `dg_xtrace` with *request* DG_XT_CONTINUE_TARGET.

In the second tracing relationship, the sequence of events required to trace a process is as follows:

1. The controlling process is created by `fork`(2) or `vfork`(2).
2. The controlling process performs a `dg_xtrace` operation with *request* DG_XT_TRACE_PID specifying the parent process's process id. If the parent process is stopped due to a job control signal (e.g., SIGSTOP) at the time DG_XT_TRACE_PID is issued, DG_XT_TRACE_PID completes normally but tracing does not actually occur until the parent process leaves the stopped state (due to a signal that continues it or terminates it).
3. The controlling process waits for the target to stop for tracing using `dg_xtrace` with *request* DG_XT_WAIT_FOR_TARGET.
4. The controller may now cause the target to continue execution using `dg_xtrace` with *request* DG_XT_CONTINUE_TARGET.

In the third tracing relationship, the sequence of events required to trace a process is as follows:

1. The controlling process performs a `dg_xtrace` operation with *request* DG_XT_TRACE_PID specifying the process id of the process to be traced. If the target is stopped due to a job control signal (e.g., SIGSTOP) at the time DG_XT_TRACE_PID is issued, DG_XT_TRACE_PID completes normally but tracing does not actually occur until the target process leaves the stopped state (due to a signal that continues it or terminates it).
2. The controlling process waits for the target to stop using `dg_xtrace` with *request* DG_XT_WAIT_FOR_TARGET.
3. The controller may now cause the target to continue execution using `dg_xtrace` with *request* DG_XT_CONTINUE_TARGET.

Tracing Requests and Macros

The *request* argument determines the precise action to be taken by `dg_xtrace`. For each *request* a macro invocation of `dg_xtrace` exists that initializes the supplied `dg_xtrace_u` union pointer with the appropriate values. Each macro name if of the form

DG_XTRACE_*name*([*arguments*])

where *name* is the same as the corresponding *name* in *request*. *arguments* includes various combinations of the following:

process-id	The process id of the target
target_addr	A byte address in the target process
controlling_buf	A byte address in the controlling process,
nchar	The number of bytes to transfer between the two processes
signal	A signal to be sent when continuing a target
wait_ptr	A pointer to an integer that receives information about the traced process
target_space	The subspace of the process's address space (Visible Address Space or User Area) to which a read/write operation is directed: DG_XT_USER_SPACE or DG_XT_ADDRESS_SPACE

The *xtrace_union_ptr* argument is the only argument from the macro interface which is passed to `dg_xtrace`. The other argument values in the interface are placed into the union at which *xtrace_union_ptr* points. (See the actual `dg_xtrace` interface above.)

Users should only access `dg_xtrace` through the macro invocations. The requests and their macros are as follows:

DG_XT_TRACE_ME

macro: `DG_XTRACE_TRACE_ME()`

In the first configuration, the target process must issue this request if it is to be traced by its controller (its parent). This operation marks the target as being traced so that it will be stopped for tracing upon receipt of a signal rather than the state specified by its signal handler. A return value of 0 is always returned with this request. [Unexpected results may ensue if the controller (in this case the parent) does not expect to trace the target. For example, the controller may not cause the target to continue after a signal. Also, the target will be terminated if the controller terminates.]

The other requests can be used only by the controller process.

DG_XT_TRACE_PID

macro: `DG_XTRACE_TRACE_PID(`*process-id*, *xtrace_union_ptr*`)`

With this request, the controlling process issues a request to establish a tracing relationship with a target process (specified by process id). This request will fail if:

- the effective-user-id of the controlling process does not match the real-user-id and saved-user-id of the target process, and the effective-group-id of the controlling process does not match the real-group-id and saved group id of the target process, OR
- the user id of the controlling process is not the superuser. If the tracing relationship is successfully established, it turns on the target's trace flag. This stipulates that the target should stop for tracing upon receipt of a signal rather than the state specified by its signal handler. Note that this request does NOT cause a signal to be sent to the target or otherwise cause the target to stop. If the relationship cannot be established the request will fail, in which case the error condition ESRCH is asserted. Under no circumstances may a process trace

itself!

DG_XT_UNTRACE_PID

macro: `DG_XTRACE_UNTRACE_PID(`*process-id*,
target_addr,
xtrace_union_ptr)

This request allows a controller to end the tracing session it has with the process specified by *process-id*. By setting *target_addr*, the controller can specify the address that the target continues from when it first begins executing again. If *target_addr* is set to 1, then the target will continue from where it was when it stopped for tracing.

DG_XT_READ_TARGET

macro: `DG_XTRACE_READ_TARGET(`*process-id*,
target_space
target_addr,
controlling_buf,
nchar,
xtrace_union_ptr)

With this request, the *nchar* bytes beginning at location *target_addr* in *target_space* of the target's address space are placed in the location *controlling_buf* in the controller's address space. This request will fail when either *target_addr* or *controlling_buf* is not a valid byte address, *nchar* bytes cannot be read or *target_addr* is not a valid address in *target_space*, in which case the error condition EIO is asserted. The request will also fail with EIO if the caller is not the controller of the target or the target is not stopped for tracing, though the stop need not have been acknowledged with DG_XTRACE_WAIT_FOR_TARGET.

DG_XT_WRITE_TARGET

macro: `DG_XTRACE_WRITE_TARGET(`*process-id*,
target_space,
target_addr,
controlling_buf,
nchar,
xtrace_union_ptr)

With this request, the *nchar* bytes beginning at location *controlling_buf* in the controller's address space are placed in the location *target_addr* in *target_space* of the target's address space. This request will fail when either *target_addr* or *controlling_buf* is not a valid byte address, *nchar* bytes cannot be written or *target_addr* is not a valid address in *target_space*, in which case the error condition EIO is asserted. The request will also fail with EIO if the caller is not the controller of the target or the target is not stopped for tracing, though the stop need not have been acknowledged with DG_XTRACE_WAIT_FOR_TARGET.

DG_XT_WAIT_FOR_TARGET

macro: DG_XTRACE_WAIT_FOR_TARGET(*process-id-ptr*,
wait_ptr,
options,
xtrace_union_ptr)

With this request, the controlling process waits for one of his traced processes. This request works like wait3(2) except that it waits only for a traced process that has received a signal (the wait3(2) WUNTRACED option is ignored). When one of the controller's targets receives a signal, this function returns the pid of that process and "acknowledges" that the target has stopped so that subsequent calls to this function will not return that same pid until the target has been continued and it receives another signal. If none of the targets have received a signal, then the caller is pended unless the WNOHANG option is specified, in which case pid 0 is returned and the caller is not pended. EIO is returned if the caller is not the controller for any traced process.

Note: When a process terminates or an untraced process receives one of the job control stop signals, the status is always reported via wait3(2) performed by the parent, never via DG_XTRACE_WAIT_FOR_TARGET. When a traced process stops for tracing, the status is always reported only to the controller process. If the controller is not the parent, then the status is only reported via DG_XTRACE_WAIT_FOR_TARGET and the traced state of the process will be invisible as far as any wait3(2) calls made by the parent. If the controller is the parent, the status is reported through either wait3(2) or DG_XTRACE_WAIT_FOR_TARGET, but not both. The receipt of signal status will be reported by whichever call is made first; but if the other call is then made the process will already have been acknowledged and will not be reported again.

DG_XT_CONTINUE_TARGET

macro: DG_XTRACE_CONTINUE_TARGET(*process-id*,
signal,
xtrace_union_ptr)

This request causes the target to resume execution. If the *signal* argument is a valid signal number, the target resumes execution as if it had incurred that signal. This request will fail if *signal* is not 0 or a valid signal number, in which case the error condition EIO is asserted. The request will also fail with EIO if the caller is not the controller of the target or the target is not stopped for tracing, though the stop need not have been acknowledged with DG_XTRACE_WAIT_FOR_TARGET.

DG_XT_TERMINATE_TARGET

macro: DG_XTRACE_TERMINATE_TARGET(*process-id*,
xtrace_union_ptr)

This request causes the target to terminate with the same consequences as exit, except that the target will not stop for tracing again as part of exit. The request will fail with EIO if the caller is not the controller of the target or the target is not stopped for tracing, though the stop need not have been acknowledged with DG_XTRACE_WAIT_FOR_TARGET.

DG_XT_INHERIT_TRACE_ON_FORK

macro: `DG_XTRACE_INHERIT_TRACE_ON_FORK(`*process-id*, *xtrace_union_ptr*)

This request sets the `inherit_trace_on_fork` flag for the target. Having this flag set will cause any processes forked by the target to be traced with the same controller.

DG_XT_SINGLESTEP_TARGET

macro: `DG_XTRACE_SINGLE_STEP_TARGET(`*process-id*, *signal*, *xtrace_union_ptr*)

This request will cause the target to continue execution for one instruction. The target will be allowed to run with the signal specified by *signal* and will take an exception after executing its next instruction. This exception will cause it to stop again for tracing. This request will fail if *signal* is not a valid signal number.

DG_XT_STOP_ON_STORE

macro: `DG_XTRACE_STOP_ON_STORE(`*process-id*, *address*, *length*, *stop_on_store_id*, *xtrace_union_ptr*)

This request causes the target to set a watch point at the memory location specified by *address* for the length in bytes specified by *length*. The address does not need to be a valid part of the target's address space. Also, a length of zero bytes is a legal parameter. This call returns to the controller a unique stop on store id. The maximum number of watch points that may be set is governed by the value of _PT_NUM_STOP_ON_STORE_IDS, which is defined in `user.h`. If this limit is exceeded, `dg_xtrace` will return EINVAL. When the target writes to an area marked for stop on store, it will report this event by sending a SIGTRAP signal without arguments to itself. Then, when the process stops to handle the signal, it will alert its controller that it has stopped and can be debugged. The controller can then read the target's address space to determine which stop on store requests were hit. The requests will be identified by a bit map returned in the DG value added structure of the ptrace_user structure. The bits corresponding to the stop on store ids that were hit will be set. See `user.h` for a complete definition of the bit map. The stop on store requests that the controller set are cleared when the controller sends the message DG_XT_REMOVE_ALL_STOP_ON_STORE to the target. This will also remove the record of which stop on store requests were hit. Because of this, the controller should read the target's user area before it clears the requests.

DG_XT_REMOVE_ALL_STOP_ON_STORE

macro: `DG_XTRACE_REMOVE_ALL_STOP_ON_STORE(`*process-id*, *xtrace_union_ptr*)

This request removes all stop on store requests. In doing so, it also removes the record of which requests have been hit. This request will always succeed, even if no stop on store requests have been made.

To forestall possible fraud, `dg_xtrace` inhibits the set-user-id facility on subsequent `exec` calls. If a traced process calls `exec`, it will stop before executing the first instruction of the new image showing signal SIGTRAP.

ACCESS CONTROL

None.

RETURN VALUE

0	Completed successfully.
−1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EIO	*Request* is an illegal number.
ESRCH	*Process-id* identifies a target that does not exist.
ESRCH	No tracing relationship can be established.
EINVAL	One of the arguments is invalid
EFAULT	One of the arguments specifies a bad address.

SEE ALSO

`exec`(2), `ptrace`(2), `signal`(2), `wait`(2).

NAME

dup – duplicate an open file descriptor

SYNOPSIS

```
int  dup (fildes)
int  fildes;
```

where:

fildes A valid, active file descriptor

DESCRIPTION

If *fildes* is a valid, active descriptor, then this call returns a new file descriptor with both descriptors sharing the same object pointer. The new descriptor is set to remain open across `exec` system calls. This call is identical to `new_filedes = fcntl (`*filedes*`, F_DUPFD, 0)`.

The new descriptor belongs to the same descriptor class as *filedes*. That is, if *filedes* is a shared descriptor, the new descriptor is in the same shared descriptor array, and if *filedes* is a per-process descriptor, then so is the new descriptor.

ACCESS CONTROL

None.

RETURN VALUE

0..*NOFILE*–1 The value of the new file descriptor.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF *Fildes* is not a valid, active descriptor.

EMFILE All descriptors are open.

SEE ALSO

`accept`(2), `close`(2), `creat`(2), `dup2`(2), `exec`(2), `fcntl`(2), `getdtablesize`(2), `open`(2), `pipe`(2), `socket`(2), `socketpair`(2), `dg_attach_to_shared_descriptors`(2).

STANDARDS

When using `m88kbcs` as the Software Development Environment target, the `dup` function will be emulated using the `fcntl`(2) system call. Since this is an emulation, a slight performance degradation may be noticed in comparison to using `dup` in `/lib/libc.a`.

NAME

dup2 – duplicate an open file descriptor onto a specific descriptor

SYNOPSIS

```
int  dup2 (old_fildes, new_fildes)
int  old_fildes;
int  new_fildes;
```

where:

old_fildes	A valid, active file descriptor
new_fildes	Another file descriptor

DESCRIPTION

Dup2 combines the functionality of the dup and close operations.

If *old_fildes* is an active, valid descriptor and *new_fildes* is a valid descriptor (active or not), *new_fildes* is made a duplicate of *old_fildes*. If *old_fildes* and *new_fildes* already refer to the same object pointer, no changes occur. In all other situations in which *new_fildes* is active, it is closed before being made a duplicate of *old_fildes*. The close-on-exec flag is set so the descriptor remains open across exec(2) operations. For a further discussion of the semantics of duplication and closing, see the dup and close operations respectively.

If *old_fildes* equals *new_fildes*, no changes occur. However, an error will be returned if *old_fildes* is not an active, valid descriptor.

Per-process descriptors are numbered from 0 to the system upper limit on per-process descriptors, MAX_PP_DESCRIPTORS. They are also bounded above by the hard and soft file descriptor limits for the calling process (see open(2)). Shared file descriptors are numbered from MAX_PP_DESCRIPTORS+1 to MAX_SHARED_DESCRIPTORS. Dup2(2) can duplicate a descriptor of one class into a descriptor of another class.

ACCESS CONTROL

None.

RETURN VALUE

new_fildes	The value of the new file descriptor given by *new_fildes*.
–1	An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EBADF	*Old_fildes* is not a valid, active descriptor.
EBADF	*New_fildes* is not a valid descriptor.

SEE ALSO

accept(2), close(2), creat(2), dup(2), exec(2), fcntl(2), getdtablesize(2), open(2), pipe(2), socket(2), socketpair(2), dg_attach_to_shared_descriptors(2).

STANDARDS

When using m88kbcs as the Software Development Environment target, the dup2 function will be emulated using BCS system calls. Since this is an emulation requiring several BCS system calls, a slight performance degradation may be noticed in comparison to using dup2 in /lib/libc.a.

NAME

exec: execl, execv, execle, execve, execlp, execvp – execute a file

SYNOPSIS

`#include <unistd.h>`

`int execl (const char *path, const char *arg0, ..., const char *argn, (char *)0);`

`int execv (const char *path, char *const *argv);`

`int execle (const char *path, const char *arg0, ..., const char *argn, (char *0), const char *envp[]);`

`int execve (const char *path, char *const *argv, char *const *envp);`

`int execlp (const char *file, const char *arg0, ..., const char *argn, (char *)0);`

`int execvp (const char *file, char *const *argv);`

where:

path A pointer to a pathname that identifies the new process file.

file A pointer to the new process file. If *file* does not contain a slash character, the path prefix for this file is obtained by a search of the directories passed in the PATH environment variable [see environ(5)]. The environment is supplied typically by the shell [see sh(1)]. If the new process file is not an executable object file, **execlp** and **execvp** use the contents of that file as standard input to sh(1).

arg (0 through *n*) Pointers to null-terminated character strings. These strings constitute the argument list available to the new process image. Minimally, *arg0* must be present. It will become the name of the process, as displayed by the ps command. Conventionally, *arg0* points to a string that is the same as *path* (or the last component of *path*). The list of argument strings is terminated by a (char *)0 argument.

argv 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 should point to a string that is the same as *path* (or its last component). *argv* is terminated by a null pointer.

envp 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**, **execv**, **execvp**, and **execlp**, the C run-time start-off routine places a pointer to the environment of the calling process in the global object extern char **environ, and it is used to pass the environment of the calling process to the new process.

DESCRIPTION

Exec in all its forms overlays a new process image on an old process. The new process image is constructed from an ordinary, executable file. This file is either an executable object file, or a file of data for an interpreter. There can be no return from a successful **exec** because the calling process image is overlaid by the new process image.

An interpreter file begins with a line of the form

#! *pathname* [*arg*]

where *pathname* is the path of the interpreter, and *arg* is an optional argument. When an interpreter file is exec'd, the system execs the specified interpreter. The pathname specified in the interpreter file is passed as *arg0* to the interpreter. If *arg* was specified in the interpreter file, it is passed as *arg1* to the interpreter. The remaining arguments to the interpreter are *arg0* through *argn* of the originally exec'd file.

When a C program is executed, it is called as follows:

```
int main (int argc, char *argv[], char *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 at least one, and the first member of the array points to a string containing the name of the file.

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 that are being caught by the calling process are set to the default disposition in the new process image [see `signal`(2)]. Otherwise, the new process image inherits the signal dispositions of the calling process.

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.

If the effective user-ID is `root` or super-user, the set-user-ID and set-group-ID bits will be honored when the process is being controlled by `ptrace`.

The shared memory segments attached to the calling process will not be attached to the new process [see `shmop`(2)].

Any user specified page locking properties [see `memcntl`(2) with the `MCL_FUTURE` option] are not inherited. In in affect, these will be reset for the new process.

Profiling is disabled for the new process; see `profil`(2).

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

nice value [see `nice`(2)]
process ID
parent process ID
process group ID
supplementary group IDs
`semadj` values [see `semop`(2)]
session ID [see `exit`(2) and `signal`(2)]
trace flag [see `ptrace`(2) request 0]
time left until an alarm clock signal [see `alarm`(2)]
current working directory
root directory
file mode creation mask [see `umask`(2)]
resource limits [see `getrlimit`(2)]
`utime`, `stime`, `cutime`, and `cstime` [see `times`(2)]
file-locks [see `fcntl`(2) and `lockf`(3C)]

controlling terminal
process signal mask [see `sigprocmask`(2)]
pending signals [see `sigpending`(2)]

Upon successful completion, `exec` marks for update the `st_atime` field of the file. Should the `exec` succeed, the process image file is considered to have been `open()`-ed. The corresponding `close()` is considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to `exec`.

`exec` will fail and return to the calling process if one or more of the following are true:

EACCES	Search permission is denied for a directory listed in the new process file's path prefix.
E2BIG	The number of bytes in the new process's argument list is greater than the system-imposed limit of 5120 bytes. The argument list limit is sum of the size of the argument list plus the size of the environment's exported shell variables.
EACCES	The new process file is not an ordinary file.
EACCES	The new process file mode denies execution permission.
EAGAIN	Total amount of system memory available when reading via raw I/O is temporarily insufficient.
EFAULT	*Path*, *argv*, or *envp* point to an illegal address.
EINTR	A signal was caught during the `exec` system call.
ELIBACC	Required shared library does not have execute permission.
ELIBEXEC	Trying to `exec`(2) a shared library directly.
ELOOP	Too many symbolic links were encountered in translating *path* or *file*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and the file system type does not allow it.
ENAMETOOLONG	The length of the *file* or *path* argument exceeds {PATH_MAX}, or the length of a *file* or *path* component exceeds {NAME_MAX} while _POSIX_NO_TRUNC is in effect.
ENOENT	One or more components of the new process pathname of the file do not exist or is a null pathname.
ENOTDIR	A component of the new process path of the file prefix is not a directory.
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.
ETXTBSY	The new process file is a pure procedure (shared text) file that is currently open for writing by some process.
ENOMEM	The new process requires more memory than is allowed by the system-imposed maximum MAXMEM.
ENOLINK	*path* points to a remote machine and the link to that machine is no longer active.

SEE ALSO

alarm(2), exit(2), fcntl(2), fork(2), getrlimit(2), memcntl(2), nice(2), ptrace(2), semop(2), signal(2), sigpending(2), sigprocmask(2), times(2), umask(2), lockf(3C), system(3S), a.out(4), environ(5).

NAME

exit, _exit – terminate process

SYNOPSIS

```
#include <stdlib.h>

void exit(int status);

#include <unistd.h>

void _exit(int status);
```

where:

status An integer indicating the status to be returned

DESCRIPTION

The functions exit() and _exit() terminate the calling process. The function exit() may cause addition processing to be done before the process exits [see atexit(3C) and *fclose*(3S)].

In addition, termination will have the following consequences:'

All of the file descriptors, directory streams and message catalogue descriptors open in the calling process are closed.

A SIGCHLD signal is sent to the calling process's parent process.

If the calling process' parent process is executing either wait(), waitpid(), or waitid() see [wait(2), waitpid(), waitid()],and has not set its SA_NOCLDWAIT flag [see sigaction(2)], it is notified of the calling process' termination, the calling process' status is made available to it, and the lifetime of the calling process ends.

If the calling process' parent process is not executing either wait(), waitpid(), or waitid(), and has not set its SA_NOCLDWAIT flag, the calling process is transformed into a zombie process. The status of the child process will be available to the parent process when the parent process subsequently executes a wait function. At that time, the lifetime of the calling process will end.

If the parent process of the calling process has set its SA_NOCLDWAIT flag, the status will be discarded, and the lifetime of the calling process will end immediately.

The parent process ID of all of the calling process' child processes is set to the process ID of a the initialization process, which has a process ID of 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.

An accounting record is written on the accounting file if the system's accounting routine is enabled [see acct(2)].

If the process is a controlling process, SIGHUP is sent to the foreground process group of its controlling terminal and its controlling terminal is deallocated.

If the calling process has any stopped children whose process group will be orphaned when the calling process exits, or if the calling process is a member of a process group that will be orphaned when the calling process exits, that process group will be sent `SIGHUP` and `SIGCONT` signals.

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)`, `semop(2)`, `sigaction(2)`, `signal(2)`, `times(2)`, `wait(2)`, `atexit(3C)`.

NOTES

See `signal(2)`.

NAME

`exportfs` – make a directory available for mounting via NFS

SYNOPSIS

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

int     exportfs (directory_name, export_entry_ptr)
char    * directory_name;
struct export * export_entry_ptr;
```

where:

directory_name — The local directory or file to be made available for mounting over NFS from NFS clients

export_entry_ptr — A pointer to a struct export that describes how this entry should be exported

DESCRIPTION

The `exportfs` system call makes a local directory (or file) available for mounting via NFS by NFS clients. The way the entry is exported is contained in the structure pointed to by *export_entry_ptr*. See `<sys/export.h>` for details. If *directory_name* has already been exported, it is logically re-exported with a new entry constructed per *export_entry_ptr*. No attempt is made to insure that either the parent of *directory_name* or a child of *directory_name* has been exported already. Such enforcment is left to the invoking code.

ACCESS CONTROL

The calling process's effective user id must be superuser.

RETURN VALUE

0 — Successful completion.

–1 — An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EPERM — The process's effective user id is not superuser; or *directory_name* contains a character not in the allowed character set.

EINVAL — The *ex_flags* field of the structure pointed to by *export_entry_ptr* was non-zero and was not EX_RDONLY or EX_RDMOSTLY; or the *ex_auth* field of the structure pointed to by *export_entry_ptr* was not AUTH_UNIX; or more than EXMAXROOTADDRS were indicated to be part of this export entry or EX_RDMOSTLY was set in the *ex_flags* field of the structure and more than EXMAXADDRS were indicated to be part of this export entry.

EFAULT — Some part of the structure pointed to by *export_entry_ptr* lies outside the process's readable address space; or *directory_name* does not completely reside in the process's address space or *directory_name* does not terminate in the process's address space.

EOPNOTSUPP — Kernel support for NFS is not present.

ENOENT — *directory_name* does not exist; or a non-terminal component of *directory_name* does not exist.

ENOTDIR	A non-terminal component of *directory_name* was not a directory or symbolic link.
ENAMETOOLONG	*directory_name* or a component of *directory_name* exceeds the length limit for pathnames.
ENOMEM	There are not enough system resources to resolve *directory_name* or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.

SEE ALSO

`exportfs`(1M), `mount`(2).

NAME

`fchdir` – change the working directory of the calling process

SYNOPSIS

```
#include <unistd.h>

int  fchdir (int fildes)
```

where:

filedes The open file descriptor of the desired working directory

DESCRIPTION

Filedes refers to a directory that is made the current working directory of the calling process. If *filedes* refers to a symbolic link, the target of the symbolic link is made the current working directory.

If the call fails, the current working directory is not changed.

ACCESS CONTROL

The calling process must have execute permission to the named directory.

RETURN VALUE

0 The current directory was successfully changed.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EACCES	Execute permission to the directory is denied.
EBADF	The *filedes* is not an open file descriptor.
ENOTDIR	The open *filedes* does not reference a directory.

SEE ALSO

`chroot`(2).

NAME

fchmod – change mode of file

SYNOPSIS

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

where:

fildes File descriptor

mode File's new mode

DESCRIPTION

Fildes is a valid, active descriptor referring to an open file of type ordinary, directory, block special, or character special, or symbolic link. The file must reside on a file system device mounted read-write. Fchmod changes the file's mode (st_mode) in a manner semantically identical to the way chmod does.

ACCESS CONTROL

The effective user id of the calling process must be superuser or match the user id of the file.

The process's effective user id must be superuser to set the sticky bit. To set the set-group-id bit, the process's effective user id must be superuser or its effective group id must match the file's group id. Failure to meet the requirements for setting one of these bits does not produce an error. Note that meeting the first access requirement is sufficient to allow a process to set the set-user-id bit.

RETURN VALUE

0 The file's mode was successfully changed.

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EBADF *Fildes* is not a valid, active file descriptor.

EINVAL The file descriptor refers to a pipe or an object that is not a file.

EPERM The process is denied permission to change the file's mode.

EROFS The named file resides on a file system device mounted read-only.

SEE ALSO

chmod(1), **chmod**(2), **chown**(2), **creat**(2), **fchown**(2), **fcntl**(2), **fstat**(2), **mknod**(2), **mknod**(2), **open**(2), **read**(2), **stat**(2), **write**(2).

NAME

`fchown` – change user id and group id of a file

SYNOPSIS

```
#include <unistd.h>

int fchown (fildes, user, group)
int fildes;
int user;
int group;
```

where:

fildes	Descriptor of the file
user	File's new user id
group	File's new group id

DESCRIPTION

Fildes is a valid, active descriptor referring to an open file of type ordinary, directory, block special, block character, or symbolic link. The file must reside on a file system device mounted read-write. `Fchown` changes the file's user id (`st_uid`) and group id (`st_gid`) to the values contained in *user* and *group*, respectively.

The semantics of changing the user and group ids are the same as those described in `chown`.

ACCESS CONTROL

The effective user id of the calling process must be superuser or match the file's user id.

RETURN VALUE

0 The user id and group id of the file were successfully changed.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	The descriptor does not refer to an open file.
EINVAL	The descriptor refers to a pipe or an object that is not a file.
EPERM	Permission to change the file's user and group ids is denied.
EROFS	The named file resides on a file system device mounted read-only.

SEE ALSO

`chgrp`(1), `chmod`(1), `chown`(1), `chmod`(2), `chown`(2), `fchmod`(2).

NAME

`fcntl` – file descriptor control

SYNOPSIS

```
#include <fcntl.h>

int  fcntl (fildes, command, argument)
int  fildes;
int  command;
int  argument;
```

where:

fildes A valid, active file descriptor

command A file control command

argument An argument, either an integer (when *command* is one of F_DUPFD, F_GETFD, F_SETFD, F_SETFL, F_GETOWN, F_SETOWN or F_CHKFL) or a pointer to a `struct flock` (when *command* is one of F_GETLK, F_SETLK, F_SETLKW or F_FREESP)

DESCRIPTION

The `fcntl` call provides a variety of operations on descriptors. *fildes* is an active, valid descriptor. *command* is a file control command to be performed on *fildes* using *argument* as an argument. Not all commands require an argument. The commands available are:

F_DUPFD The first (lowest numbered) inactive descriptor of the calling process, greater than or equal to *argument*, is made a duplicate of *fildes*. Thus, both descriptors refer to the same object pointer. The new descriptor's `close-on-exec` attribute is set to remain open across `exec` operations.

This operation is equivalent to `dup` if *argument* is zero and `dup2` if *argument* is an inactive descriptor.

F_SETFD Set the `close-on-exec` attribute of *fildes* to the low-order bit of *argument*. If the low-order bit is 0, the file will remain open across `exec` operations; otherwise, the file will be closed upon execution of the `exec` operation.

F_GETFD Return the `close-on-exec` attribute of *fildes*.

F_SETOWN Invoke the type manager of the object to which *fildes* refers to set the process id or process group id receiving the SIGIO and SIGURG signals for the object. Process group ids are specified by supplying *argument* as negative, otherwise *argument* is interpreted as a process id.

F_GETOWN Query the type manager of the object to which *fildes* refers for the process id or process group id currently receiving SIGIO and SIGURG signals for the object. Process group ids are returned as negative values.

F_SETFL Invoke the type manager of the object to which *fildes* refers to set the object pointer status flag bits to *argument*. Only the following flag bits may be set: O_NONBLOCK, O_NDELAY, O_APPEND, O_SYNC, and O_ASYNC.

F_GETFL Query the type manager of the object to which *fildes* refers for the object pointer status flag. The following flags will be returned if set in the object pointer status flag: O_NONBLOCK, O_NDELAY, O_APPEND, O_SYNC, O_ASYNC, O_RDWR, O_RDONLY, and O_WRONLY.

F_SETLK Set or clear a file lock according to *argument*, which is interpreted to be a pointer to a `struct flock`. F_SETLK is used to set read locks, set write locks, or remove either type of lock. If a read or a write lock cannot be set, `fcntl` returns immediately with a value of −1.

F_SETLKW This command is the same as F_SETLK except if a read or write lock is blocked by other locks, the process will pend until the segment to be locked is free.

F_GETLK Get the first lock that blocks the lock description specified by *argument*, which is interpreted to be a pointer to a `struct flock`. The information retrieved overwrites the information passed to `fcntl` in *argument*. If no lock is found that would prevent this lock from being set, *argument* is unchanged, except for the lock type, which is set to F_UNLCK.

F_CHKFL Check *argument* to see if it would be valid if passed as the argument to a F_SETFL `fcntl` command. Return 0 if valid, −1 if not.

F_FREESP Free storage space associated with the file descriptor according to the `struct flock` pointed to by *argument*. The portion to be freed is specified by *l_whence*, *l_start* and *l_len*. If the *l_len* field is 0, the file is truncated. Currently, the only supported operation is truncation (that is, *l_len* must be 0).

ACCESS CONTROL

None.

RETURN VALUE

The value returned may be one of the following regardless of the value of *command*:

−1 An error occurred. `errno` is set to indicate the error.

If *command* is F_GETFD, the value returned may be one of the following:

0 or 1 Completed successfully. The value of the `close-on-exec` flag is returned.

If *command* is F_SETFD, F_SETFL, F_CHKFL, F_FREESP or F_SETOWN:

0 Completed successfully.

If *command* is F_DUPFD, the value returned may be one of the following:

argument..NOFILE−1
Completed successfully. A new file descriptor is returned.

If *command* is F_GETOWN, the value returned may be one of the following:

owner Completed successfully. If the value is negative, *−owner* is the process group id returned. Otherwise *owner* is a process id. Note that the process group may be 1, in which case, −1 will be returned.

If *command* is F_GETFL, the value returned may be one of the following:

file_flags Completed successfully. The value of the file flags is returned.

If *command* is F_GETLK, F_SETLK, or F_SETLKW, the value returned may be the following:

0 Completed successfully.

DIAGNOSTICS

`Errno` may be set to one of the following error codes regardless of the value of *command*:

EBADF *Fildes* is not a valid, active descriptor.

EINVAL *Command* is not one of the known values; or *argument* is not a valid descriptor; or the `flock` structure pointed to by *argument* is outside the process's readable address space.

If *command* is F_DUPFD, `errno` may be set to one of these values:

EMFILE All descriptors are currently open.

If *command* is F_SETLK, F_SETLKW, or F_GETLK, `errno` may be set to one of these values:

EBADF The caller requested a read lock, and the channel does not provide read access, or the caller requested a write lock and the channel does not provide write access.

EINTR The command was F_SETLKW and the process was interrupted while pending on a lock.

EDEADLK The command was F_SETLKW and a deadlock would exist if the lock were granted.

EACCES The command was F_SETLK and the type of lock sought is a read lock (F_RDLCK) or write lock (F_WRLCK), and the segment of a file to be locked is already write-locked by another process, or the type is a write lock and the segment of a file to be locked is already read-locked or write-locked by another process.

Additional errors may be given by the type managers.

SEE ALSO

`close(2)`, `creat(2)`, `dup(2)`, `dup2(2)`, `exec(2)`, `fork(2)`, `getdtablesize(2)`, `open(2)`, `pipe(2)`, `sigvec(2)`, `socket(2)`, `socketpair(2)`, `fcntl(5)`.

NAME

`fetch_and_add` – indivisible fetch and add to memory location

SYNOPSIS

```
tb0 0,r0,401
```

DESCRIPTION

`Fetch_and_add` is a extended operation (XOP) that indivisibly fetches the value of a user memory location and adds to that memory location.

Input registers are:

`r2` Address of 32 bit user memory location to be fetched and added to. This address must be aligned on a 4 byte boundary.

`r3` 32 bit integer to add to the user memory location.

Return registers are:

`r1` Unchanged

`r2` Unchanged

`r3` Unchanged

`r4` Undefined

`r5` New value of the memory location

`r6` Undefined

`r7` Status: 0 means success (memory location was set to the new value), 1 means some fault occurred when accessing the memory location.

`r8` Old value of the memory location

`r9` Undefined

`r10` through `r31`
Unchanged

The value of the memory location pointed to by `r2` is read, the value in `r3` is added to it using unsigned arithmetic (no overflow exceptions are generated), and the result is stored back into the same memory location. The old and new values of the memory location are returned. If any fault (including a page fault) occurs when accessing the memory location, an error code is returned and the memory location is not modified.

The `fetch_and_add` XOP executes indivisibly with respect to all other `fetch_and_add` operations running on any processor in the system that may be going on simultaneously to the same physical memory location. It does not necessarily execute indivisibly with respect to `fetch_and_add` operations to other memory locations, or with respect to other XOPs to the same memory location, or with respect to normal loads and stores or I/O traffic to the memory location.

While the XOP is being executed, the user process will not be descheduled, will not page fault, and will not be terminated. If a fault of any kind (page fault, protection fault, misaligned access fault, for example) occurs when the XOP references user data, the XOP terminates and returns an error. User code is responsible for catching the error, touching the data item so that the fault can be handled, and then retrying the XOP. The execution time of the XOP is charged to user mode, not kernel mode. User profiling ticks that occur while the XOP is in progress are accounted to the instruction following the trap instruction.

Fetch_and_add must be invoked with an assembly language trap instruction. Typically the trap instruction is done from an assembly language routine that is linked with the application and called as a standard subroutine in the high level language in which the application is written.

EXAMPLE

```
                global _fetch_and_add

; routine is entered with the memory address in r2 and the
; amount to add in r3.  The following "C" statement invokes
; this routine correctly:
;
;       int    location, amount, old_value;
;
;       old_value = fetch_and_add(&location, amount);
;
;

_fetch_and_add:
        tb0    0,r0,401      ; trap to the fetch-and-add xop (#401)
        bcnd   ne0,r7,_fault      ; had a data access fault
        jmp.n  r1            ; back to the caller
        or     r2,r0,r8      ; old value is function return value

; if a data access fault occurred during the XOP, control will
; come here.  A data access fault could just be a page fault,
; or it could be a real error such as a protection violation.
; Hence we do a simple load of the memory location so that
; whatever the fault is, it will occur on the load.  If it is
; a page fault, the page fault will be handled by bring the
; page into memory.  If it is a protection fault, an
; appropriate signal will be sent.  If the load succeeds (as
; on a page fault), then we try the XOP over again.
_fault:
        ld   r8,r0,r2        ; read memory location.  We really
                             ; don't care care where the data goes.
                             ; r8 is convenient.
        br   _fetch_and_add; try the XOP again
```

Note that the above routine is just an example. Applications can and should modify the routine to get exactly the desired interface. For example, the new value can be returned instead of the old value by moving r5 instead of r8 into r2.

SEE ALSO

store_conditional(2).

NAME

fork – create a new process

SYNOPSIS

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

DESCRIPTION

Fork creates a new process with its own address space that is initialized to the contents of the calling process's address space at the time the fork call is made. The new process is entered into the process tree as a child of the calling process.

The following attributes in the new process are set to the values the parent process had at the time of the fork call:

Environment
Signal handling settings
(i.e., SIG_DFL, SIG_IGN, function address)
Real- and effective-user-id
Real- and effective-group-id
Tty group id
Group list
Profiling on/off status
Nice value (see nice)
All attached shared memory segments (see shmat)
Process group ID
Current working directory
Root directory
File mode creation mask (see umask)
Resources utilization limits (see ulimit, setrlimit)
Controlling terminal device
Close on exec flag
Attached shared descriptor array

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 (the process ID of its parent).
- The child process has its own copy of each of the parent's per-process object descriptors, with the close-on-exec flag in each set to the value from the corresponding object descriptor in the parent. Each of the child's object descriptors shares a common object pointer with the corresponding object descriptor of the parent.
- File locks set by the parent are not inherited by the child.
- The set of signals pending for the child process is cleared.
- All semaphore adjustment values are cleared [see semop(2)].
- Process locks, text locks, data locks, and locks on any other regions of the parent process's address space are not inherited by the child [see plock(2) and memcntl(2)].
- The child process's current resources consumed and cumulative resources consumed by its children are set to zero. This includes the child's utime, stime, cutime, and cstime [see setrlimit(2)].

- The value of ITIMER_REAL (used by alarm and setitimer) is set to 0 so that SIGALRMs are disabled. ITIMER_VIRTUAL and ITIMER_PROF (used by setitimer) are also set to 0 (ie. all pending alarms are cleared in the child).
- Unless specifically set, the child process does not inherit tracing [see `ptrace`(2) and `dg_xtrace`(2)].

ACCESS CONTROL

If the new process would cause the system-imposed limit on the total number of processes in the system to be reached, an error is returned and the new process is not created, unless the calling process has an effective-user-id of 0. In other words, only the superuser is allowed to create a process that causes the limit to be reached.

RETURN VALUE

Upon successful completion, `fork` returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of −1 is returned to the parent process, no child process is created, and `errno` is set to indicate the error.

DIAGNOSTICS

`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.

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

ENOMEM The process requires more memory than the system is able to supply.

SEE ALSO

`dg_xtrace`(2), `exec`(2), `memcntl`(2), `nice`(2), `plock`(2), `ptrace`(2), `semop`(2), `shmat`(2), `signal`(2), `sigset`(2), `times`(2), `ulimit`(2), `umask`(2), `vfork`(2), `wait`(2).

NAME

`fstat` – get file status

SYNOPSIS

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

int     fstat (fildes, buffer_ptr)
int     fildes;
struct stat * buffer_ptr;
```

where:

fildes	A valid, active file descriptor
buffer_ptr	Address of a `stat` buffer to fill

DESCRIPTION

`Fstat` returns the current attributes of the file referenced by *fildes* into the status buffer at the location specified by *buffer_ptr*.

The interpretation of the file's attributes depends on the file's type [see `stat(5)` for details]. The subject file must be of type 'ordinary-disk-file', 'directory', 'block-special-file', 'character-special-file', 'fifo-special-file' 'pipe', or 'socket'.

If fstat fails, the contents of the stat buffer are undefined.

ACCESS CONTROL

Read, write, or execute permission of the file is not required. However, for *fildes* to be active, the file must be open for reading or writing.

RETURN VALUE

0 The `fstat` operation was successful.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	*Fildes* is not a valid, active file descriptor.
EFAULT	*buffer_pointer* points to an invalid address.

SEE ALSO

`chmod(2)`, `chown(2)`, `creat(2)`, `dg_mstat(2)`, `fchmod(2)`, `fchown(2)`, `link(2)`, `lstat(2)`, `mknod(2)`, `pipe(2)`, `read(2)`, `stat(2)`, `time(2)`, `unlink(2)`, `utime(2)`, `utimes(2)`, `write(2)`.

NAME

`fstatfs` – get information about a mounted file system

SYNOPSIS

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

int     fstatfs (fildes, statfs_buffer, len, fstype)
int     fildes;
struct statfs * statfs_buffer;
int     len;
int     fstype;
```

where:

filedes	File descriptor for any file within the file system to be reported on
statfs_buffer	A `statfs` structure where information about the file system is returned
len	Length of the user's buffer
fstype	Type of the file system [see `statfs(2)`]

DESCRIPTION

Fildes is a valid, active descriptor referring to an open file of any type (ordinary, directory, FIFO, block special, character special, or symbolic link). Terminal symbolic links are resolved in the system call that returned *filedes*. `Fstatfs` returns the same information about the mounted file system that contains the file that `statfs` does.

ACCESS CONTROL

None.

RETURN VALUE

0	The file system information was successfully returned.
–1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	*Fildes* is not a valid, active file descriptor.
EFAULT	Some part of the statfs structure pointed to by *statfs_buffer* lies outside of the process's writable address space.
EINVAL	*Fildes* refers to a pipe or socket.

SEE ALSO

`chmod(2)`, `chown(2)`, `creat(2)`, `fchmod(2)`, `fchown(2)`, `link(2)`, `mknod(2)`, `pipe(2)`, `read(2)`, `statfs(2)`, `time(2)`, `times(2)`, `ustat(2)`, `write(2)`, `fs(4)`, `statfs(5)`.

NAME

`fstatvfs` – return information about a file system

SYNOPSIS

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

int  fstatvfs (int fildes, struct statvfs *buffer)
```

where:

filedes The open file descriptor of any file in the file system to be reported on.

buffer Address of a statvfs buffer where file system information will be returned

DESCRIPTION

Filedes is a valid, active descriptor refering to an open file of any type. The information returned concerns details about file system where the *filedes* resides and is the same as that of statvfs:

```
ulong f_bsize;    /* file system block size */
ulong f_frsize;   /* file system fragment size */
ulong f_blocks;   /* total number of blocks of f_frsize
                     contained in the file system */
ulong f_bfree;    /* total number of free blocks */
ulong f_bavail;   /* number of free blocks available to
                     the non-super-user */
ulong f_fsid;     /* file system identifier */
char  f_basetype[FSTYPSZ]; /* null-terminated fs type
                              name */
ulong f_flag;     /* bit mask of flags */
ulong f_namemax;  /* maximum file name length */
char  f_fstr[32]; /* file system specific string */
```

f_basetype contains the file system type name and is null-terminated. The value for the constant FSTYPSZ is defined in the <statvfs.h> file.

The f_flag can return the following:

```
ST_RDONLY     /* a read-only file system */
ST_NOSUID     /* file system does not support the
                 setuid or setgid semantics */
```

ACCESS CONTROL

None.

RETURN VALUE

0 The information was successfully returned in the statvfs buffer.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF The *filedes* is not an open file descriptor.

SEE ALSO

`chmod`(2), `chown`*(2)*, `create`*(2)*, `dup`*(2)*, `fcntl`*(2)*, `link`*(2)*, `mknod`*(2)*, `open`*(2)*, `pipe`*(2)*, `read`*(2)*, `time`*(2)*, `unlink`*(2)*, `ustat`*(2)*, `utime`*(2)*, `write`*(2)*.

NAME

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

SYNOPSIS

```
#include <unistd.h>

int  fsync (fildes)
int  fildes;
```

where:

fildes A valid, active file descriptor that refers to an open file.

DESCRIPTION

The `fsync` system call causes all modified data and attributes of the file to be written to disk. Write operations performed by the `write` or `writev` system calls are atomic, so it is not possible for `fsync` to record the results of a partial write. Also, while the `fsync` is being performed, further writes to the file are blocked. Thus, `fsync` ensures that a snapshot of the file's state is on physical disk.

Upon successful completion of all writes, the file's time of last file attribute change (`st_ctime`) is set to the current time.

ACCESS CONTROL

None.

RETURN VALUE

0 The synchronization was successful.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	*Fildes* is not a valid descriptor.
EINVAL	*Fildes* refers to an object other than a file.
EIO	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

sync(1M), sync(2).

NAME

`ftruncate` – truncate a file

SYNOPSIS

```
#include <unistd.h>

int   ftruncate (fildes, length)
int   fildes;
long  length;
```

where:

fildes A valid, active file descriptor

length Maximum length of file after truncation

DESCRIPTION

This call is similar to the `truncate` system call, except that `ftruncate` takes a file descriptor instead of a pathname to identify the file.

Otherwise, the semantics of this call are identical to those of truncate.

ACCESS CONTROL

Fildes must be open for writing.

RETURN VALUE

0 The file was successfully truncated.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	*Fildes* is not a valid, active descriptor.
EINVAL	*Fildes* is not open for writing.
EINVAL	*Fildes* references a socket, not a file.
EINTR	`Fcntl` was interrupted while waiting for a lock.

SEE ALSO

`creat(2), open(2), truncate(2).`

NAME

getcontext, setcontext – get and set current user context

SYNOPSIS

```
#include <ucontext.h>

int getcontext(ucontext_t *ucp);

int setcontext(ucontext_t *ucp);
```

where:

ucp The name of a structure to contain user context information

DESCRIPTION

These functions are useful for implementing user level context switching between multiple threads of control within a process.

getcontext initializes the structure pointed to by *ucp* to the current user context of the calling process. The user context is defined by ucontext(5) and includes the contents of the calling process's machine registers, signal mask and execution stack.

setcontext restores the user context pointed to by *ucp*. The call to setcontext does not return; program execution resumes at the point specified by the context structure passed to setcontext. The context structure should have been one created either by a prior call to getcontext or passed as the third argument to a signal handler [see sigaction(2)]. If the context structure was one created with getcontext, program execution continues as if the corresponding call of getcontext had just returned.

ACCESS CONTROL

No access checking is performed.

RETURN VALUE

On successful completion, setcontext does not return and getcontext returns 0. Otherwise, a value of -1 is returned and errno is set to indicate the error.

DIAGNOSTICS

Under the following conditions, getcontext and setcontext fail and set errno to:

EFAULT *ucp* points to an invalid address.

NOTES

When a signal handler is executed, the current user context is saved and a new context is created by the kernel. If the process leaves the signal handler via longjmp(3C) the original context will not be restored, and future calls to getcontext will not be reliable. Signal handlers should use siglongjmp(3C) or setcontext instead.

SEE ALSO

sigaction(2), sigaltstack(2), sigprocmask(2), ucontext(5).

NAME

getdents – get directory entries in a filesystem-independent format

SYNOPSIS

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

int  getdents (fildes, buffer, nbyte)
int  fildes;
char *buffer;
unsigned nbyte;
```

where:

fildes File descriptor of the directory to list

buffer Buffer to hold the directory entries

nbyte Size of the buffer in bytes

DESCRIPTION

Getdents attempts to read *nbyte* bytes from the directory associated with *fildes* and then format them as filesystem-independent directory entries in the buffer pointed to by *buffer*. Since the filesystem-independent directory entries are of variable length, in most cases the actual number of bytes returned will be strictly less than *nbyte*.

The filesystem entry is specified by the dirent structure. The dirent structure is defined as

```
struct dirent {
        long            d_ino;
        off_t           d_off;
        unsigned short  d_reclen;
        char            d_name[1];
};
```

The d_ino entry is a number that is unique for each file in the filesystem. The d_off is the offset of that directory entry in the actual filesystem directory. The field d_name is the beginning of the character array giving the name of the directory entry. This name is null terminated and may have at most MAXNAMLEN characters. The variable length of filenames makes the file system independent variable length. The value d_reclen is the record length of this entry.

On devices capable of seeking, getdents starts at a position in the file given by the file pointer associated with *fildes*.

Upon return, the actual number of bytes transferred is returned. The current position pointer associated with *fildes* is set to point to the next block of entries.

ACCESS CONTROL

None.

RETURN VALUE

>0 The number of bytes actually transferred.

0 The end of the directory has been reached.

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EBADF Fildes is not a valid file descriptor for reading.

EFAULT	Buf points outside the allocated address space.
EINVAL	nbyte is not large enough for one directory entry.
ENOENT	The current file pointer for the directory is not located at a valid entry.
ENOTDIR	Fildes is not a directory.
EIO	An I/O error occurred while accessing the file system.

SEE ALSO

`lseek`(2), `open`(2), `dirent`(4).

NAME

getdomainname – get name of current domain

SYNOPSIS

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

where:

name Buffer to receive domain name

namelen Buffer length in bytes

DESCRIPTION

Getdomainname returns the name of the network domain of the host system, as previously set by setdomainname. The parameter *namelen* specifies the size of the name array. The returned name is null-terminated unless insufficient space is provided.

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

Domain names are limited to MAXDOMAINNAMELEN characters, which is defined in <user/param.h>.

Calling getdomainname before calling setdomainname produces undefined results.

ACCESS CONTROL

None.

RETURN VALUE

0 Completed successfully.

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EFAULT The *name* parameter gave an invalid address, or the *namelen* parameter specified a length less than zero.

SEE ALSO

gethostid(2), gethostname(2), setdomainname(2).

NAME
`getdtablesize` – return the number of open files the current process can have

SYNOPSIS
```
int getdtablesize ()
```

DESCRIPTION
The `getdtablesize` call returns the number of open files the current process can have. This can be changed via the *setrlimit*(2) and *ulimit*(2) system calls.

ACCESS CONTROL
None.

RETURN VALUE
NOFILE   The number of open files that the current process may have.

DIAGNOSTICS
None.

SEE ALSO
`close`(2), `dup`(2), `fcntl`(2), `open`(2), `select`(2), `setrlimit`(2), `ulimit`(2).

NAME

getegid – get the effective-group-id

SYNOPSIS

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

gid_t   getegid ()
```

DESCRIPTION

Getegid returns the effective-group-id of the calling process.

ACCESS CONTROL

No access checking is performed.

RETURN VALUE

0..MAXUID The return value is always the effective-group-id of the calling process.

DIAGNOSTICS

None.

SEE ALSO

getuid(2), geteuid(2), getgid(2), setuid(2), setgid(2), setregid(2), setreuid(2).

NAME

geteuid – get the effective-user-id

SYNOPSIS

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

uid_t   geteuid ()
```

DESCRIPTION

Geteuid returns the effective-user-id of the calling process.

ACCESS CONTROL

No access checking is performed.

RETURN VALUE

0..MAXUID The return value is always the effective-user-id of the calling process.

DIAGNOSTICS

None.

SEE ALSO

getuid(2), getgid(2), getegid(2), setuid(2), setgid(2), setregid(2), setreuid(2).

NAME

getfh – return the file handle of the export entry containing filename

SYNOPSIS

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

int  getfh (filename, filehandle_ptr)
char * filename;
fhandle_t * filehandle_ptr;
```

where:

filename	The filename to get the filehandle of
filehandle_ptr	Where to put filehandle for the file specified by *filename*

DESCRIPTION

If *filename* has been exported via exportfs(2), then the filehandle for the filesystem that is exported is returned. This system call is normally used only by the NFS mount daemon.

ACCESS CONTROL

The calling process's effective user id must be superuser.

RETURN VALUE

0	Successful completion. The file handle for *descriptor* is returned in *filehandle_ptr*.
–1	An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EINVAL	No export entry exists for this filename.
EFAULT	Some part of the space pointed to by *filehandle_ptr* lies outside the process's readable address space.
EPERM	The caller is not superuser.
EOPNOTSUPP	Kernel support for NFS is not present.

SEE ALSO

dg_mount(2).

NAME

getgid – get the real-group-id

SYNOPSIS

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

uid_t   getgid ()
```

DESCRIPTION

Getgid returns the real-group-id of the calling process.

ACCESS CONTROL

No access checking is performed.

RETURN VALUE

0..MAXUID The return value is always the real-group-id of the calling process.

DIAGNOSTICS

None.

SEE ALSO

getuid(2), geteuid(2), getegid(2), setuid(2), setgid(2), setregid(2), setreuid(2).

NAME

getgroups, setgroups – get or set supplementary group access list IDs

SYNOPSIS

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

int getgroups(int gidsetsize, gid_t *grouplist)

int setgroups(int ngroups, const gid_t *grouplist)
```

where:

gidsetsize The number of entries currently in *grouplist*
grouplist An array of group IDs
ngroups The number of entries to be in *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*.

setgroups sets the supplementary group access list of the calling process from the array of group IDs specified by *grouplist*. The number of entries is specified by *ngroups* and can not be greater than {NGROUPS_MAX}. This function may be invoked only by the super-user.

RETURN VALUE

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.

DIAGNOSTICS

getgroups will fail if:

EINVAL The value of *gidsetsize* is non-zero and less than the number of supplementary group IDs set for the calling process.

setgroups will fail if:

EINVAL The value of *ngroups* is greater than {NGROUPS_MAX}.

EPERM The effective user ID is not super-user.

Either call will fail if:

EFAULT A referenced part of the array pointed to by *grouplist* is outside of the allocated address space of the process. See groups(1) in the *User's Reference Manual*.

SEE ALSO

chown(2), getuid(2), setuid(2),

NAME

gethostid – get unique identifier of current host

SYNOPSIS

```
long gethostid ()
```

DESCRIPTION

Gethostid returns the 32-bit identifier for the host system.

Calling gethostid before calling sethostid produces undefined results.

ACCESS CONTROL

None. (Call always succeeds.)

RETURN VALUE

hostid Completed successfully.

DIAGNOSTICS

None.

SEE ALSO

getdomainname(2), gethostname(2), sethostid(2).

NAME

gethostname – get name of current host

SYNOPSIS

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

where:

name Buffer to receive hostname

namelen Buffer length in bytes

DESCRIPTION

`Gethostname` returns the standard hostname for the host system, as previously set by `sethostname`. The parameter *namelen* specifies the size of the *name* string. The returned name is null-terminated unless insufficient space is provided. Insufficient space will truncate the name.

Hostnames are limited to MAXHOSTNAMELEN characters, which is defined in `<sys/param.h>`.

Calling `gethostname` before calling `sethostname` returns a zero-length hostname.

ACCESS CONTROL

None.

RETURN VALUE

0 Completed successfully.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to the following error code:

EFAULT The *name* parameter gave an invalid address, or the *namelen* parameter specified a length less than zero.

SEE ALSO

`getdomainname`(2), `gethostid`(2), `sethostname`(2).

NAME

getitimer, setitimer – get or set value of interval timer

SYNOPSIS

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

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

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

where:

which ITIMER_REAL, ITIMER_VIRTUAL, or ITIMER_PROF
value Name of pointer to structure for storing timer value
ovalue Name of pointer to structure for storing old timer value

DESCRIPTION

The system provides each process with three interval timers, defined in sys/time.h. The getitimer call stores the current value of the timer specified by *which* into the structure pointed to by *value*. The setitimer call sets the value of the timer specified by *which* to the value specified in the structure pointed to by *value*, and if *ovalue* is not NULL, stores the previous value of the timer in the structure pointed to by *ovalue*.

A timer value is defined by the itimerval structure [see gettimeofday(3C) for the definition of timeval], which includes the following members:

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

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

Time values smaller than the resolution of the system clock are rounded up to this resolution.

The three timers are:

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

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

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

RETURN VALUE

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

DIAGNOSTICS

Under the following conditions, the functions getitimer and setitimer fail and set errno to:

EFAULT *value* or *ovalue* specified a bad address

EINVAL The specified number of seconds is greater than 100,000,000, the number of microseconds is greater than or equal to 1,000,000, or the *which* parameter is unrecognized.

SEE ALSO

`alarm`(2), `gettimeofday`(2).

NOTES

The microseconds field should not be equal to or greater than one second.

`setitimer` is independent of the `alarm` system call.

Do not use `setitimer` with the `sleep` routine. A `sleep` following a `setitimer` wipes out knowledge of the user signal handler.

NAME

getmsg, getpmsg – get a message from a stream

SYNOPSIS

```
#include <stropts.h>

int getmsg (filedes, control_info_ptr, data_info_ptr, flags_ptr)
int filedes;
struct strbuf * control_info_ptr;
struct strbuf * data_info_ptr;
int * flags_ptr;

int getpmsg (filedes, control_info_ptr, data_info_ptr, band_ptr, flags_ptr)
int filedes;
struct strbuf * control_info_ptr;
struct strbuf * data_info_ptr;
int * band_ptr;
int * flags_ptr;
```

where:

filedes	File descriptor
control_info_ptr	A pointer to a structure describing the control buffer or NULL, if there is no control buffer
data_info_ptr	A pointer to a structure describing the data buffer or NULL, if there is no data buffer
band_ptr	On input, specifies the minimum band message to retrieve. On output, the band of the message retrieved.
flags_ptr	On input, the type of message desired; on output, the type of message retrieved

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.

The function getpmsg does the same thing as getmsg, but provides finer control over the priority of the messages received. Except where noted, all information pertaining to getmsg also pertains to getpmsg.

filedes specifies a file descriptor referencing an open stream. *control_info_ptr* and *data_info_ptr* each point to a strbuf structure, which contains the following members:

buf Pointer to the first byte of the control or data buffer.

maxlen The maximum size, in bytes, of the buffer or a negative number, if information of that type is not requested.

len Ignored on input. On output, contains the number of bytes of control or data information placed in the buffer or –1, if there is no information of that type present in the message or that type of information was not requested. This field is valid on output only if the status OK is returned.

The control buffer is used to hold the control part of the message (those message blocks before the first block of type M_DATA; typically either M_PCPROTO or M_PROTO blocks). The data buffer is used to hold the data part of the message (any blocks after and including the first M_DATA block). If a strbuf pointer is NULL or "maxlen" is negative, the corresponding part of the message is not processed and is left on the stream. If the control (or data) "maxlen" is 0 and the first control (or data) block has a data buffer of length 0, that block is removed from the message and the control (or data) "len" is set to 0. If "maxlen" is 0 and the first block of the corresponding type has a non-zero buffer, however, the block is left on the message and "len" is set to 0. If "maxlen" is less than the length of the corresponding portion of the message, "maxlen" bytes are retrieved and placed in the caller's buffer. The remainder of the message is left on the stream and a non-zero return value is provided as described under `RETURN VALUE`.

By default, `getmsg` processes the first available message on the stream head read queue. However, a user may choose to retrieve only high priority messages by setting the integer pointed by *flags_ptr* to `RS_HIPRI`. In this case, `getmsg` processes the next message only if it is a high priority message. If the integer pointed by *flags_ptr* is 0, `getmsg` retrieves any message available on the stream head read queue. In this case, on return, the integer pointed to by *flags_ptr* will be set to if a high priority message was retrieved, or 0 otherwise.

For `getpmsg`, the flags are different. *flags_ptr* points to a bitmask with the following mutually-exclusive flags defined: `MSG_HIPRI`, `MSG_BAND`, and `MSG_ANY`. Like `getmsg`, `getpmsg` processes the first available message on the stream head read queue. A user may choose to retrieve only high-priority messages by setting the integer pointed to by *flags_ptr* to `MSG_HIPRI` and the integer pointed to by *band_ptr* to 0. In this case, `getpmsg` will only process the next message if it is a high-priority message. In a similar manner, a user may choose to retrieve a message from a particular priority band by setting the integer pointed to by *flags_ptr* to `MSG_BAND` and the integer pointed to by *band_ptr* to the priority band of interest. In this case, `getpmsg` will only process the next message if it is in a priority band equal to, or greater than, the integer pointed to by *band_ptr*, or if it is a high-priority message. If a user just wants to get the first message off the queue, the integer pointed to by *flags_ptr* should be set to `MSG_ANY` and the integer pointed to by *band_ptr* should be set to 0. On return, if the message retrieved was a high-priority message, the integer pointed to by *flags_ptr* will be set to `MSG_HIPRI` and the integer pointed to by *band_ptr* will be set to 0. Otherwise, the integer pointed to by *flags_ptr* will be set to `MSG_BAND` and the integer pointed to by *band_ptr* will be set to the priority band of the message.

If `O_NDELAY` and `O_NONBLOCK` are clear, `getmsg` blocks until a message of the type specified by *flagsp* is available on the stream head read queue. If `O_NDELAY` or `O_NONBLOCK` has been set and a message of the specified type 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` continues to operate normally, as described above, until the stream head read queue is empty. Thereafter, it returns 0 in the `len` fields of *ctlptr* and *dataptr*.

ACCESS CONTROL

Fildes must be open for reading.

RETURN VALUE

−1	An error occurred. `errno` is set to indicate the error.
0	An entire message was successfully read.

MORECTL	A message was partially read. More control information is waiting for retrieval.
MOREDATA	A message was partially read. More data information is waiting for retrieval.
MORECTL | MOREDATA	A message was partially read. More control and data information is waiting for retrieval.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EAGAIN	The O_NDELAY or O_NONBLOCK flag was set and a message of the requested type was not available.
EBADF	*Fildes* is not a valid, active descriptor open for reading.
EBADMSG	The message at the head of the stream is not a type that is retrievable by `getmsg`(2).
EFAULT	The arguments pointed to by *control_info_ptr*, *data_info_ptr*, and *flags_ptr* do not lie entirely within the caller's readable and writable address space.
EINTR	A signal was caught during the `getmsg` call.
EINVAL	An illegal value was specified by **flags_ptr*
EINVAL	The stream referred to by *fildes* is linked under a multiplexor.
ENOSTR	*Fildes* does not refer to a stream.

SEE ALSO

`putmsg`(2).

NOTE

The user should avoid using `O_NDELAY` and instead use `O_NONBLOCK`.

NAME

`getpagesize` – get the system page size

SYNOPSIS

```
int getpagesize()
```

DESCRIPTION

The `getpagesize()` function returns the number of bytes in a page. Some memory management calls require knowledge of this page size. The page size is a system page size and may not be the same as the underlying hardware page size.

ACCESS CONTROL

No access check is made.

RETURN VALUE

The `getpagesize()` function returns the system page size, in bytes.

DIAGNOSTICS

No errors are returned.

SEE ALSO

`sysconf`(2).

NAME

getpeername – get name of connected peer

SYNOPSIS

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

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

where:

s	File descriptor of socket whose name is requested
name	Structure to receive the name of connected peer
namelen	On input contains the number of bytes available for the peer name; updated to indicate the number of bytes returned

DESCRIPTION

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

ACCESS CONTROL

None. (See domain information [inet(3N), unix_ipc(6F)] for domain-specific restrictions.)

RETURN VALUE

0	Completed successfully.
-1	An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EBADF	The argument *s* is not a valid descriptor.
ENOTSOCK	The argument *s* is not a file of type S_IFSOCK (socket special).
ENOTCONN	The socket is not connected.
ENOBUFS	Insufficient resources were available in the system to perform the operation.
EFAULT	The *name* parameter points to memory not in a valid part of the process address space, or the *namelen* parameter is < 0.

SEE ALSO

bind(2), connect(2), getsockname(2), socket(2), inet(3N), unix_ipc(6F).

NAME

getpgrp – get process group ID

SYNOPSIS

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

gid_t  getpgrp ()
```

DESCRIPTION

The getpgrp() function returns the process group ID of the calling process.

RETURN VALUE

See DESCRIPTION.

DIAGNOSTICS

The getpgrp() function is always successful, and no return value is reserved to indicate an error.

SEE ALSO

setpgid(2), setpgrp(2), setsid(2), sigaction(2)

COPYRIGHTS

NAME

`getpgrp2` – get process group

SYNOPSIS

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

gid_t  getpgrp2 (pid)
pid_t  pid;
```

where:

pid The process whose process group is to be returned. If zero, the process group of the calling process is returned.

DESCRIPTION

The process group of the specified process is returned by `getpgrp2`. If *pid* is zero the process group of the calling process is returned.

ACCESS CONTROL

No access checking is performed.

RETURN VALUE

process group id The call succeeded. The process group id is returned.

–1 The specified process does not exist. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to the following error code:

ESRCH The process specified by *pid* does not exist.

SEE ALSO

`getpgrp`(2), `setpgrp`(2), `setpgrp2`(2).

NAME

getpid, getpgrp, getppid, getpgid – get process, process group, and parent process IDs

SYNOPSIS

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

pid_t getpid(void);

pid_t getpgrp(void);

pid_t getppid(void);

pid_t getpgid(pid_t pid);
```

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.

getpgid returns the process group ID of the process whose process ID is equal to *pid*, or the process group ID of the calling process, if *pid* is equal to zero.

RETURN VALUE

getpid, getpgrp, and getppid return the values given above.

Upon successful completion, getpgid returns a process group ID. Otherwise, a value of (pid_t) −1 is returned and errno is set to indicate the error.

DIAGNOSTICS

getpid, getpgrp, and getppid always succeed.

getpgid will fail if one or more of the following is true:

EPERM	The process whose process ID is equal to *pid* is not in the same session as the calling process, and the implementation does not allow access to the process group ID of that process from the calling process.
ESRCH	There is no process with a process ID equal to *pid*.

SEE ALSO

exec(2), fork(2), getpid(2), getsid(2), intro(2), setpgid(2), setsid(2) setpgrp(2), signal(2).

NAME

getppid – get parent process-id

SYNOPSIS

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

pid_t  getppid ()
```

DESCRIPTION

Getppid returns the process-id of the calling process's parent.

ACCESS CONTROL

No access checking is performed.

RETURN VALUE

process-id   The process-id of the calling process's parent is always returned.

DIAGNOSTICS

None.

SEE ALSO

exec(2), fork(2), setpgrp(2), signal(2).

NAME

`getpriority` – get process scheduling priority

SYNOPSIS

```
#include &lt;sys/resource.h&gt;

int  getpriority (which, who)
int  which;
int  who;
```

where:

which How the argument *who* is to be interpreted: PRIO_PROCESS, PRIO_PGRP, or PRIO_USER

who One or more process IDs, process group IDs, or user IDs, depending on the value of *which*

DESCRIPTION

One or more processes are identified by the combination of the arguments *which* and *who*. If *which* is PRIO_PROCESS, *who* is interpreted as a process ID and a single process identified. If *which* is PRIO_PGRP, *who* is interpreted as a process group ID, and all processes that are members of that group are identified. If *which* is PRIO_USER, *who* is interpreted as a user ID, and all processes with effective-user-id of *who* are identified.

A *who* value of 0 is interpreted as the calling process's process ID, process group ID, and effective-user-id, respectively, for the three cases listed. For example, all processes in the calling process' process group may be identified with *which* set to PRIO_PGRP and *who* set to zero.

The `getpriority` call returns the highest priority (lowest numerical value) enjoyed by any of the identified processes.

ACCESS CONTROL

No access checking is performed.

RETURN VALUE

If no errors occur, `getpriority` returns the highest priority (lowest numerical value) enjoyed by any of the identified processes. If an error occurs, –1 is returned and `errno` is set to identify the error.

Since `getpriority` can legitimately return the value –1, it is necessary to clear the external variable `errno` prior to the call, then check it afterward to determine if a –1 is an error or a legitimate value.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

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

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

SEE ALSO

`fork`(2), `nice`(2).

NAME

getpsr – return the current contents of the processor status register

SYNOPSIS

```
#include <sys/m88kbcs.h>

unsigned int  getpsr ()
```

DESCRIPTION

The getpsr system call returns the processor status register for the calling process.

ACCESS CONTROL

No access checking is performed.

RETURN VALUE

processor status register
: The processor status register of the calling process.

DIAGNOSTICS

None.

SEE ALSO

setpsr(2).

NAME

`getrlimit, setrlimit` – control maximum system resource consumption

SYNOPSIS

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

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

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

where:

resource The resource for which the limits are to be returned or set.

rlp A pointer to a structure into which the limit values are to be placed or read from

DESCRIPTION

Limits on the consumption of a variety of system resources by a process and each process it creates may be obtained with `getrlimit` and set with `setrlimit`.

Each call to either `getrlimit` or `setrlimit` identifies a specific resource to be operated upon as well as a resource limit. A resource limit is a pair of values: one specifying the current (soft) limit, the other a maximum (hard) limit. Soft limits may be changed by a process to any value that is less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or equal to the soft limit. Only a process with an effective user ID or superuser can raise a hard limit. Both hard and soft limits can be changed in a single call to `setrlimit` subject to the constraints described above. Limits may have an infinite value of `RLIM_INFINITY`. *rlp* is a pointer to `struct rlimit` that includes the following members:

```
rlim_t   rlim_cur;   /* current (soft) limit */
rlim_t   rlim_max;   /* hard limit */
```

`rlim_t` is an arithmetic data type to which objects of type `int`, `size_t`, and `off_t` can be cast without loss of information.

The possible resources, their descriptions, and the actions taken when current limit is exceeded, are summarized in the table below:

Resources	Description	Action
`RLIMIT_CORE`	The maximum size of a core file in bytes that may be created by a process. A limit of 0 will prevent the creation of a core file.	The writing of a core file will terminate at this size.
`RLIMIT_CPU`	The maximum amount of CPU time in seconds used by a process.	`SIGXCPU` is sent to the process. If the process is holding or ignoring `SIGXCPU`, the behavior is scheduling class defined.
`RLIMIT_DATA`	The maximum size of a process's heap in bytes.	`brk`(2) will fail with errno set to `ENOMEM`.

Resources	Description	Action
RLIMIT_FSIZE	The maximum size of a file in bytes that may be created by a process. A limit of 0 will prevent the creation of a file.	SIGXFSZ is sent to the process. If the process is holding or ignoring SIGXFSZ, continued attempts to increase the size of a file beyond the limit will fail with errno set to EFBIG.
RLIMIT_NOFILE	The maximum number of open file descriptors that the process can have.	Functions that create new file descriptors will fail with errno set to EMFILE. Or, when attempting an fcntl() "F_DUPFD", return EINVAL.
RLIMIT_STACK	The maximum size of a process's stack in bytes. The system will not automatically grow the stack beyond this limit.	SIGSEGV is sent to the process. If the process is holding or ignoring SIGSEGV, or is catching SIGSEGV and has not made arrangements to use an alternate stack [see sigaltstack(2)], the disposition of SIGSEGV will be set to SIG_DFL before it is sent.
RLIMIT_AS	The maximum size of a process's mapped address space in bytes. This same resource may be accessed by setting *resource* to RLIMIT_VMEM.	brk(2) and mmap(2) functions will fail with errno set to ENOMEM. Also, attempting an exec(2) on an image greater than this limit will fail, setting errno to ENOMEM.
RLIMIT_RSS	The maximum size in bytes that a process' resident set size my grow to.	

Because limit information is stored in the per-process information, the shell builtin ulimit must directly execute this system call if it is to affect all future processes created by the shell.

The value of the current limit of the following resources affect these implementation defined constants:

Limit	Implementation Defined Constant
RLIMIT_FSIZE	FCHR_MAX
RLIMIT_NOFILE	OPEN_MAX

RETURN VALUE

Upon successful completion, the function getrlimit returns a value of 0; otherwise, it returns a value of -1 and sets errno to indicate an error.

DIAGNOSTICS

Under the following conditions, the functions getrlimit and setrlimit fail and set errno to:

EINVAL if an invalid *resource* was specified; or in a setrlimit call, the new rlim_cur exceeds the new rlim_max.

EPERM if the limit specified to `setrlimit` would have raised the maximum limit value, and the caller is the superuser

SEE ALSO

`open`(2), `sbrk`(2), `sigaltstack`(2), `ulimit`(2), `malloc`(3C), `signal`(5).

NAME

getrusage – get information about resource utilization

SYNOPSIS

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

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

where:

who RUSAGE_SELF and RUSAGE_CHILDREN, identifying whether to return information about the calling process or about the calling process's acknowledged terminated children

rusage A pointer to an area in the calling process's address space where the resource usage information is to be written

DESCRIPTION

Getrusage returns information describing the resources utilized by the current process, or the sum of the resources utilized by each of its acknowledged terminated children, depending on the value of *who*. The rusage structure pointed to by *rusage* is filled in with the information. See the description of the *rusage* structure for the details of each field.

If an error occurs, *rusage* is unmodified.

ACCESS CONTROL

The argument *rusage* must point to an area in the calling process's address space that is valid and has write access.

RETURN VALUE

0 Successful completion.

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EINVAL The *who* argument was not RUSAGE_SELF or RUSAGE_CHILDREN.

EFAULT The *rusage* argument specifies an invalid area of the calling process's address space or an area which does not have read/write access.

SEE ALSO

gettimeofday(2), wait(2).

NAME

getsid – get session ID

SYNOPSIS

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

pid_t getsid (pid_t pid)
```

where:

pid A process identifier

DESCRIPTION

The function getsid returns the session ID of the process whose process ID is equal to *pid*. If *pid* is equal to (pid_t)0, getsid returns the session ID of the calling process.

RETURN VALUE

Upon successful completion, the function getsid returns the session ID of the specified process; otherwise, it returns a value of (pid_t)-1 and sets errno to indicate an error.

ACCESS CONTROL

No access checking is performed.

DIAGNOSTICS

Under the following conditions, the function getsid fails and sets errno to:

ESRCH if there is no process with a process ID equal to *pid*.

SEE ALSO

exec(2), fork(2), getpid(2), setpgid(2), setsid(2).

NAME

getsockname – get socket name

SYNOPSIS

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

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

where:

s	File descriptor of socket whose name is requested
name	Structure to receive the socket name
namelen	On input contains the number of bytes available for the name; updated to indicate the number of bytes returned

DESCRIPTION

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

ACCESS CONTROL

None. See domain specific information [`unix_ipc`(6F) and `inet`(3N)] for restrictions per domain.

RETURN VALUE

0	Completed successfully.
–1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	The argument *s* is not an active valid descriptor.
ENOTSOCK	The argument *s* is not a file of type S_IFSOCK (socket special).
ENOBUFS	Insufficient resources were available in the system to perform the operation.
EFAULT	The *name* parameter points to memory not in a valid part of the process address space, or the *namelen* parameter is < 0.

SEE ALSO

`bind`(2), `socket`(2), `inet`(3N), `unix_ipc`(6F).

NAME

getsockopt – get options on a socket

SYNOPSIS

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

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

where:

s	File descriptor of socket to get options from
level	Level in socket that the options apply to
optname	Name of option to return (options are defined in socket.h)
optval	Buffer to receive options
optlen	On input contains the number of bytes available for the options; updated to indicate the number of bytes returned

DESCRIPTION

Getsockopt retrieves options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost socket level.

When getting socket options, the caller must specify the level at which the option resides and the name of the option. To retrieve options at the socket level, *level* is specified as SOL_SOCKET. To get options at any other level, the protocol number of the appropriate protocol controlling the option is supplied. Consult domain specific documentation for more information related to a specific protocol.

The parameters *optval* and *optlen* identify a buffer in which the value for the requested option(s) are to be returned. *optlen* is a value/result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be returned, *optval* may be supplied as 0. If the buffer isn't large enough for the options, they will be truncated.

Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file <sys/socket.h> contains definitions for socket level options; see socket. Options at other protocol levels vary in format and name; consult the related documentation for the domain of the socket.

ACCESS CONTROL

SOL_SOCKET has no access restrictions. (See domain-specific documentation for any domain restrictions.)

RETURN VALUE

0 Completed successfully.

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EBADF	The argument *s* is not an active valid descriptor.
ENOTSOCK	The argument *s* is not a file of type S_IFSOCK (socket special).

ENOPROTOOPT	The option is unknown.
EFAULT	The *optval* is not in a valid part of the process address space, or *optlen* < 0.
ENOBUFS	No internal buffers available.

SEE ALSO

setsockopt(2), socket(2), inet(3N), inet(6F), tcp(6P), udp(6P), unix_ipc(6F).

NAME

gettimeofday – get date and time

SYNOPSIS

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

int gettimeofday (time_value, time_zone)
struct timeval * time_value;
struct timezone * time_zone;
```

where:

time_value	Address of a structure that will be set to the current time.
time_zone	NULL or address of a structure that will be set to the current time zone.

DESCRIPTION

`Gettimeofday` returns the system's notion of the current Greenwich time and the current time zone to the structures at the locations specified by *time_value* and *time_zone*.

If *time_zone* is NULL, the current time zone is not returned.

The time value returned is Greenwich time expressed in seconds and microseconds since midnight January 1, 1970.

The local time zone is expressed in minutes of time westward from Greenwich (*tz_minuteswest*), and a value (*tz_dsttime*) that indicates the type of daylight savings time that applies locally during the appropriate part of the year. The daylight savings time correction flag (*tz_dsttime*) further indicates the type of daylight savings time correction to apply. The accepted values are:

DST_NONE	DST does not apply.
DST_USA	USA DST correction.
DST_AUST	Australian DST correction.
DST_WET	Western European DST correction.
DST_MET	Middle European DST correction.
DST_EET	Eastern European DST correction.

The current local time may be computed using the current time zone by the following calculation:

```
local_usec = time_value->tv_usec;
local_sec = time_value->tv_sec - time_zone->tz_minuteswest * 60
                + (is_dst(time_value,time_zone) ? 60 * 60 : 0);
```

where `is_dst(tv,tz)` is some function that returns TRUE if daylight savings time is currently in effect.

ACCESS CONTROL

None.

RETURN VALUE

0 Completed successfully.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EFAULT An argument address referenced invalid memory.

SEE ALSO

`date`(1), `settimeofday`(2), `ctime`(3C).

NAME

getuid – get the real-user-id

SYNOPSIS

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

uid_t   getuid ()
```

DESCRIPTION

Getuid returns the real-user-id of the calling process.

ACCESS CONTROL

No access checking is performed.

RETURN VALUE

0..MAXUID   The return value is always the real-user-id of the calling process.

DIAGNOSTICS

None.

SEE ALSO

geteuid(2), getgid(2), getegid(2), setuid(2), setgid(2), setregid(2), setreuid(2).

NAME

`ioctl` – control a device

SYNOPSIS

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

int ioctl (fildes, command, argument)
int fildes;
int command;
int argument;
```

where:

fildes A valid, active file descriptor

command A device control command

argument A pointer to the argument for the control command

DESCRIPTION

`Ioctl` provides a variety of operations on descriptors. *Fildes* is an active, valid descriptor. *Command* is an I/O control command to be performed on *fildes* using *argument* as an argument. Not all commands require an argument.

The 88open BCS version of `ioctl` accepts values for *command* only as specified in the BCS.

The following two commands apply to any open file. In both cases, *argument* is ignored.

FIOCLEX Set the 'close-on-exec' attribute of *fildes*. This causes the file to be closed upon execution of the exec operation.

FIONCLEX Clear the 'close-on-exec' attribute of *fildes*. This causes the file to remain open across `exec` operations.

All other commands invoke the type manager of the object to which *fildes* refers to perform the I/O control operation. Usually, the object must be a character-special device.

ACCESS CONTROL

None.

RETURN VALUE

0 Completed successfully.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF *Fildes* is not a valid, active descriptor.

ENOTTY *Fildes* does not refer to a character-special device and *command* only operates on character-special devices.

EINVAL *Command* or *Argument* is not valid.

EINTR The call was interrupted by a signal.

Additional errors may be given by the type managers.

SEE ALSO

`stty`(1), `exec`(2), `fcntl`(2), `socket`(2), `termio`(7).

NAME

`kill` – send a signal to a process

SYNOPSIS

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

int kill (pid, sig)
pid_t pid;
int sig;
```

where:

pid An integer (positive, negative, or zero) indicating a process or a group of processes to be sent the signal

sig A signal number that is either one from the list given in `<signal.h>` or zero

DESCRIPTION

The `kill()` function sends a specified signal to a specified process or group of processes. If *sig* is zero (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of *pid*.

For a process to have permission to send a signal to a process designated by *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 sending process has appropriate privileges. If {_POSIX_SAVED_IDS} is defined, the saved set-user-ID of the receiving process shall be checked in place of its effective user ID. If a receiving process's effective user ID has been altered through use of the S_ISUID mode bit (see `<sys/stat.h>`), the implementation may still permit the application to receive a signal sent by the parent process or by a process with the same real user ID.

If *pid* is greater than zero, *sig* shall be sent to the process whose process ID is equal to *pid*.

If *pid* is zero, *sig* shall be sent to all processes (excluding an implementation-defined set of system processes) whose process group ID is equal to the process group ID of the sender, and for which the process has permission to send a signal.

If *pid* is −1, the behavior of the `kill()` function is unspecified.

If *pid* is negative, but not −1, *sig* shall be sent to all processes whose process group ID is equal to the absolute value of *pid*, and for which the process has permission to send a signal.

If the value of *pid* causes *sig* to be generated for the sending process, and if *sig* is not blocked, either *sig* or at least one pending unblocked signal shall be delivered to the sending process before the `kill()` function returns.

If the implementation supports the SIGCONT signal, the user ID tests described above shall not be applied when sending SIGCONT to a process that is a member of the same session as the sending process.

An implementation that provides extended security controls may impose further implementation-defined restrictions on the sending of signals, including the null signal. In particular, the system may deny the existence of some or all of the processes specified by *pid*.

The `kill()` function is successful if the process has permission to send *sig* to any of the processes specified by *pid*. If the `kill()` function fails, no signal shall be sent.

RETURN VALUE

0 Successful completion.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

If any of the following conditions occur, the `kill()` function shall return −1 and set `errno` to the corresponding value:

EINVAL The value of the *sig* argument is an invalid or unsupported signal number.

EPERM The process does not have permission to send the signal to any receiving process.

ESRCH No process or process group can be found corresponding to that specified by *pid*.

SEE ALSO

`getpid(2)`, `setsid(2)`, `sigaction(2)`, `<signal.h>`

COPYRIGHTS

STANDARDS

Since _POSIX_SAVED_IDS is defined in the DG/UX System, the saved set-user-ID of the receiving process shall be checked in place of it effective user ID.

If a receiving process's effective user ID has been altered through the use of the S_ISUID mode bit, the system permits the application to receive a signal sent by the parent process or by a process with the same real user ID.

If *pid* is −1, the behavior of `kill()` depends on the effective user ID of the calling process. If that ID is 0 (superuser), the signal will be sent to all processes except for the initialization process (PID 1). Otherwise, the signal will be sent to all processes (again excluding `init`) whose real user ID is equal to the effective user ID of the sending process.

There is one special system process that is unaffected by all calls of the form `kill(0,` *sig*). This is the system initialization process, which has PID 1.

The SIGCONT signal is supported.

The DG/UX System does not provide extended security controls, so no further restrictions are placed on the sending of signals.

The signal SIGKILL cannot be successfully sent to the system initialization process (PID 1).

NAME

`killpg` – send signal to a process or a process group

SYNOPSIS

```
int  killpg (pgrp, signal_number)
int  pgrp;
int  signal_number;
```

where:

pgrp Process-group-id of the processes being sent the signal

signal_number Type of signal being sent

DESCRIPTION

`Killpg` sends the signal *signal_number* to all processes in the process group identified by *pgrp*.

The sending process must have permission to send a signal to the process group members. The signal is sent to all those processes for which the caller has permission.

The process group identified by *pgrp* falls into one of four categories depending on the value of *pgrp*:

pgrp > 0 Signal all processes in a specified process group.

Signal_number will be sent to all processes in the process group whose process-group-id is equal to *pgrp*. System processes are never selected.

pgrp = 0 Signal all processes in the sender's process group.

Signal_number will be sent to all processes, excluding system processes, whose process-group-id is equal to the process-group-id of the sender. It is an error for the process-group-id of the sender to be zero.

pgrp = −1 Signal all processes.

If the effective-user-id of the sender is super-user, *signal_number* is sent to all processes excluding system processes. Otherwise, *signal_number* is sent to all processes, excluding system processes, whose process-group-id is −1 (i.e., no processes will be sent *signal_number*).

pgrp < −1 Signal all processes in a specified process group.

Signal_number will be sent to all processes, excluding system processes, whose process-group-id is equal to *pgrp*. [This selects no processes.]

ACCESS CONTROL

Permission to send a signal is granted in three ways:

- The sending and receiving processes have the same effective-user-id.
- The sending process is the super-user.
- The sending process is an ancestor of the receiving process and the signal being sent is SIGCONT.

RETURN VALUE

0 Completed successfully.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EINVAL	*Signal_number* is not a valid signal number.
EINVAL	*pgrp* is zero and the caller's process-group-id is zero.
ESRCH	No process can be found in the process group identified by *pgrp*.
EPERM	The sending process does not have permission to signal all members of the specified process group. This error code is not set by the Berkeley implementations.

SEE ALSO

csh(1), kill(1), kill(2), signal(2), jobs(3C).

NAME

link – create a new link to a file

SYNOPSIS

```
int  link (old_path, new_path)
char * old_path;
char * new_path;
```

where:

old_path Address of a pathname to an existing file

new_path Address of a pathname to be added

DESCRIPTION

Link adds the pathname pointed to by *new_path* to the filesystem. The file named by *new_path* is made to identify the same file as that named by *old_path*. Terminal symbolic links are not followed for *new_path*, but are followed for *old_path*.

The subject file must be of type 'ordinary-disk-file', 'block-special-file', 'character-special-file', or 'fifo-special-file'.

It is illegal to link to a file of type 'directory', 'control point directory', 'socket', or 'symbolic link'.

If link fails, no changes are made. Otherwise, the following changes are made to the subject file:

- The subject file's link count attribute (st_nlink) is incremented.
- The subject file's time of last attribute change (st_ctime) is set to the current time.

ACCESS CONTROL

The calling process must have permission to resolve *old_path* and *new_path*.

The calling process must have write permission to the directory containing the entry to be added.

RETURN VALUE

0 The new path was successfully created.

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

ENOENT	The file named by *old_path* does not exist.
EEXIST	The link named by *new_path* exists.
EPERM	Permission to link to the given file type is denied to the calling process.
EXDEV	The link named by *new_path* and the file named by *old_path* are on different file system devices.
EACCES	The requested link requires writing in a directory with a mode that denies write permission.
EROFS	The requested link requires writing in a directory on a file system device mounted read-only.
EMLINK	Too many links to one file. There can only be MAXLINK links to one file. See <sys/param.h>.

ENOSPC	No more contiguous space for file space or inodes.
ENOENT	A non-terminal component of the *old_path* or *new_path* does not exist.
ENOTDIR	A non-terminal component of the *old_path* or *new_path* was not a directory or symbolic link.
ENAMETOOLONG	*old_path* or *new_path* exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the *old_path* or *new_path* exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve *old_path* or *new_path* or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	*old_path* or *new_path* contains a character not in the allowed character set.
EFAULT	*old_path* or *new_path* does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

`symlink(2)`, `unlink(2)`, `stat(5)`.

NAME

`listen` – listen for connections on a socket

SYNOPSIS

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

where:

s File descriptor of socket to listen on.

backlog Maximum number of waiting connections.

DESCRIPTION

To accept connections, a socket is first created with `socket()`, a backlog for incoming connections is specified with `listen()`, and then the connections are accepted with `accept()`. The `listen` call applies only to sockets of type SOCK_STREAM or SOCK_PKTSTREAM.

The *backlog* parameter defines the maximum length to which the queue of pending connections may grow. If a connection request arrives with the queue full, the client will receive the error ECONNREFUSED.

In the DG/UX system the backlog is currently limited to SOMAXCONN. If a backlog greater than SOMAXCONN is specified, the backlog will be set SOMAXCONN; however no error notification will be returned. SOMAXCONN is defined in `<sys/socket.h>`.

ACCESS CONTROL

None. (See domain-specific restrictions in related documentation.)

RETURN VALUE

0 Completed successfully.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	The argument *s* is not an active valid descriptor.
ENOTSOCK	The argument *s* is not a socket.
EOPNOTSUPP	The socket is not of a type that supports the operation listen.
EINVAL	Invalid argument.

SEE ALSO

`accept(2)`, `connect(2)`, `socket(2)`, `inet(3N)`, `inet(6F)`, `unix_ipc(6F)`.

NAME

`lseek` – change object pointer's current position

SYNOPSIS

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

off_t  lseek (fildes, offset, whence)
int    fildes;
off_t  offset;
int    whence;
```

where:

fildes	A valid, active file descriptor
offset	The new position of the file pointer
whence	A value that changes the interpretation of *offset* (see **DESCRIPTION**)

DESCRIPTION

If *fildes* is a valid, active descriptor that refers to an object pointer having a current position attribute, the object pointer's current position is modified according to the *offset* and *whence* parameters as follows:

> If *whence* is SEEK_SET, the current position is set to *offset* bytes.
>
> If *whence* is SEEK_CUR, the current position is set to its current location plus *offset*.
>
> If *whence* is SEEK_END, the current position is set to the size of the object plus *offset*. The size of character special files and block special files is always zero. Hence, this option is equivalent to *whence* being SEEK_SET for these files.
>
> If *whence* is equal to any other value, the user is sent the signal SIGSYS and `errno` returns EINVAL.

It is an error for the new current position attribute to be negative.

If `lseek` fails, the object pointer is not changed.

ACCESS CONTROL

None.

RETURN VALUE

position	Completed successfully. The object pointer's new position is returned.
−1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	*Fildes* is not a valid, active descriptor.
ESPIPE	*Fildes* is associated with a pipe or a socket.
EINVAL with SIGSYS signal	*Whence* is not SEEK_SET, SEEK_CUR, or SEEK_END.
EINVAL	The resulting file pointer would be negative.

SEE ALSO

`creat`(2), `dup`(2), `dup2`(2), `fcntl`(2), `open`(2).

NAME

`lstat` – get file status

SYNOPSIS

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

int   lstat (path, buffer_ptr)
char * path;
struct stat * buffer_ptr;
```

where:

path	Address of a pathname
buffer_ptr	Address of a stat buffer to fill

DESCRIPTION

`Lstat` returns the current attributes of the symbolic link or file named by the pathname pointed to by *path* into the `stat` buffer at the location specified by *buffer_ptr*. `lstat` is equivalent to `stat` except that it will return the file attributes of a symbolic link, instead of the file attributes of the target of the symbolic link.

The interpretation of the file's attributes depend on the file's type [see `stat(5)` for details]. The subject file must be of type 'ordinary-disk-file', 'directory', 'control point directory', 'block-special-file', 'character-special-file', 'fifo-special-file', or 'symbolic-link'.

If `lstat` fails, the contents of the `stat` buffer are undefined.

ACCESS CONTROL

Read, write, or execute permission of the named file is not required, but the process must have permission to resolve *path*.

RETURN VALUE

0	The lstat operation was successful.
−1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EFAULT	*Status_buffer* points to an invalid address.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.

EFAULT — The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

chmod(2), chown(2), creat(2), dg_mstat(2), fchmod(2), fchown(2), fstat(2), link(2), mknod(2), pipe(2), read(2), stat(2), time(2), unlink(2), utime(2), utimes(2), write(2), stat(5).

NAME

`memcntl` – memory management control

SYNOPSIS

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

int memcntl(caddr_t addr, size_t len, int cmd, caddr_t arg,
            int attr, int mask);
```

where:

addr	Starting address of the target region
len	Length in bytes of the target region
cmd	Operation to perform on the target region
arg	Optional command arguments
attr	Optional page selection criteria
mask	Reserved for future use. Must be `0`.

DESCRIPTION

The function `memcntl` allows the calling process to apply a variety of control operations over some or all of its address space. For most of these operations, the affected portion of the address space is constrained to the address range [*addr*, *addr* + *len*).

The parameter *addr* must be a page aligned address. Note that the system page size can be obtained by calling either `getpagesize`(2) or `sysconf`(2) with the `_SC_PAGESIZE` parameter; both calls return identical values.

The *len* parameter need not be a multiple of the system page size. However, the implementation of `memcntl()` internally rounds *len* up to the next page size multiple to determine the address range for the target region. All further references to *len* refer to this rounded quantity.

The control operation is specified by *cmd*. All of the following values are legal:

`MC_SYNC`	Write back any modified pages in the target region to their backing storage, and optionally purge pages in the target region from primary memory.
`MC_LOCK`	Lock pages of the target region into primary memory.
`MC_UNLOCK`	Unlock pages of the target region from primary memory.
`MC_LOCKAS`	Lock the caller's entire address space into primary memory.
`MC_UNLOCKAS`	Unlock the caller's entire address space from primary memory.

The value of the *arg* parameter may affect the operation specified by *cmd*. Such behavior is defined individually for each possible value of *cmd* and is described further below. The descriptions of the remaining parameters apply equally for all legal values of *cmd*.

The scope of the control operations can be further constrained with selection criteria regarding the properties of the appropriate address range. The bit patterns of the *attr* parameter define these additional criteria, which include page protections and mapping type. If the operation is desired for all pages in the target address range, then the additional selection criteria should be disabled by setting *attr* to `0`.

If *attr* is not 0, then the operation will affect only pages whose page protections and mapping type exactly match those specified by *attr*. The mapping type is selected by specifying one of the following values in *attr*:

SHARED Select pages mapped shared, e.g., shared memory segments and file pages mapped using MAP_SHARED.

PRIVATE Select pages mapped private, e.g., text, data, and stack segments.

If *attr* is not 0, but neither of these values is specified, then only private pages are selected.

The desired page protection selection criterion is formed by ORing zero or more of the following protection flags together with the mapping type selection in *attr*. Only pages whose page protection exactly matches the specified combination of protection flags will be selected.

PROT_READ Page can be read.

PROT_WRITE Page can be written.

PROT_EXEC Page can be executed.

The following shorthand selection criteria are provided for the default page protections associated with a program's initial memory segments. One of these may be ORed with the mapping type instead of using the individual protection flags.

PROC_TEXT Text segment's default page protection.

PROC_DATA Data and stack segments' default page protection.

The *mask* parameter is reserved for future use and must have the value 0.

Each section below describes in detail the specific operations which may be applied by memcntl() to the process's address space.

MC_SYNC

Write to backing storage locations all modified pages in the range which satisfy the selection criteria given by *attr*. Optionally, purge all pages in the range with attributes *attr* from primary memory. The backing storage for a modified file page mapped using MAP_SHARED is the file's backing storage. The backing storage for all other modified pages is within the system's swap areas.

The *arg* parameter is used to alter the behavior of the operation, and may include the following flags, ORed together:

MS_ASYNC Do not wait for writebacks to complete.

MS_SYNC Wait for writebacks to complete.

MS_INVALIDATE Purge pages from primary memory.

When MS_ASYNC is specified, the call returns immediately after all required write operations are scheduled; with MS_SYNC the call will not return until all required write operations have completed. Only one of these flags should be specified; if neither is specified, MS_SYNC is assumed.

If MS_INVALIDATE is specified, then all selected pages which are memory resident will be purged from primary memory. Subsequent accesses to those pages will fault and cause the pages to be read in from backing storage. The operation will fail if any of the selected pages are locked into primary memory.

The `MS_INVALIDATE` option overrides the standard page replacement algorithms used by the system; it should be used with caution, as the system has knowledge of paging demands system-wide, while the application does not. Application and system performance may suffer when this option is used.

Note that modified pages whose backing storage is within the swap areas will not necessarily be written to backing storage unless `MS_INVALIDATE` is specified. Also, modified pages which are locked into memory will not necessarily be written to backing storage.

All addresses specified by the interval [*addr*, *addr* + *len*) must be mapped within the caller's address space. All addresses in the range which satisfy the optional selection criteria specified by the *attr* parameter will be processed.

Note that the system will write modified file pages back to the file store periodically, so use of `MC_SYNC` for that purpose is rarely needed unless the application needs to confirm that its modifications have been written to stable storage. There is a system configuration variable used to control the maximum age a modified file page can reach before the system writes it back.

`MC_LOCK`

This operation makes memory resident and locks into primary memory all pages in the range [*addr*, *addr* + *len*) which satisfy the additional selection criteria in *attr* (if any). Locked pages are forced to stay memory resident, i.e., they cannot be purged from primary memory.

All addresses specified by the interval [*addr*, *addr* + *len*) must be mapped within the caller's address space. All addresses in the range which satisfy the optional selection criteria specified by the *attr* parameter will be processed.

The *arg* parameter is reserved for future use, and must be `0`.

A given page may be locked multiple times through one mapping or multiple mappings (e.g., by multiple processes) of that page. However, within a given mapping, a single unlock operation on a page will negate the effect of all previous lock operations on that page.

The specified range should not include any mapped file pages which lie beyond the end of the file. Calls to `mmap(2)` may succeed which map pages beyond the end of the file, but attempts to access or lock such pages will fail.

Page locks are removed either explicitly, via `memcntl()`, or implicitly, when the locked mappings are destroyed, e.g., by `_exit`(2), `exec`(2), or `munmap`(2). Locks established with the lock operations are not inherited by a child process during `fork`(2).

Locking a significant fraction of primary memory may negatively affect system performance. Therefore, the system enforces a configurable limit on the total number of primary memory pages that may be locked at any time.

If a page locking operation fails, part of the operation may have been completed before the failure, possibly locking some pages which were not locked prior to the call.

`MC_LOCKAS`

This operation makes memory resident and locks into primary memory all mapped pages in the address space which satisfy the selection criteria in *attr* and *arg*.

The *addr* and *len* parameters are unused, and must both be `0`. The *arg* parameter is used to specify selection criteria, and must contain one or both of the following flags:

`MCL_CURRENT` Lock all current mappings in the caller's address space which satisfy the selection criteria in *attr* (if any).

`MCL_FUTURE` Lock all future mappings in the caller's address space, regardless of the selection criteria in *attr*. Future mappings include future extensions of the data and stack segments, as well as future mappings established via `mmap`(2) and `shmat`(2). However, future locking is reset when a process replaces its address space using `exec`(2). Once future locking has been established, there is no way to clear this state for the address space except to invoke the `MC_UNLOCKAS` operation, which will also unlock all currently locked pages.

Please refer to above discussion of `MC_LOCK` to understand the remaining functional details of page locking.

`MC_UNLOCK`

This operation removes page locks on all pages in the range [*addr*, *addr* + *len*) which satisfy the additional selection criteria in *attr* (if any).

The treatment of all other parameters besides *cmd* is identical to that of `MC_LOCK`. Please refer to the discussion above of that operation.

Performing this operation on unlocked pages will not cause an error to occur. Also, a given page may be locked multiple times through one mapping or multiple mappings (e.g., by multiple processes) of that page. However, within a given mapping, a single unlock operation on a page will negate the effect of all previous lock operations on that page.

If this operation fails, part of it may have been completed before the failure, possibly unlocking some pages which were locked prior to the call.

`MC_UNLOCKAS`

This operation removes page locks on all locked pages in the caller's address space, and resets the future locking attribute of the address space.

A given page may be locked multiple times through one mapping or multiple mappings (e.g., by multiple processes) of that page. However, within a given mapping, a single unlock operation on a page will negate the effect of all previous lock operations on that page.

The *addr*, *len*, and *arg* parameters are unused, and must all be `0`. If the parameters are correct, this operation will not fail.

ACCESS CONTROL

Any user process may invoke an `MC_SYNC` operation.

Due to their impact on system resources, all the other operations require the caller's effective user id to be superuser.

RETURN VALUE

Upon successful completion, `memcntl()` returns the value `0`. Otherwise it returns `-1`, and sets `errno` to indicate an error.

DIAGNOSTICS

Under the following conditions, the `memcntl()` fails and sets `errno` to:

EINVAL	if the *mask* parameter is not 0.
EINVAL	if the *addr* parameter is not a page aligned address.
EINVAL	if the *attr* parameter contains invalid selection criteria.
EINVAL	if MC_LOCK, MC_UNLOCK, or MC_UNLOCKAS is specified and the *arg* parameter is not 0.
EINVAL	if MC_LOCKAS is specified and the *arg* parameter is 0 or contains flags other than MCL_FUTURE or MCL_CURRENT.
EINVAL	if MC_LOCKAS or MC_UNLOCKAS is specified and the *addr* parameter is not 0.
EINVAL	if MC_LOCKAS or MC_UNLOCKAS is specified and the *len* parameter is not 0.
ENOMEM	if MC_SYNC, MC_LOCK, or MC_UNLOCK is specified and the *len* parameter is 0.
ENOMEM	if MC_SYNC, MC_LOCK, or MC_UNLOCK is specified and some address in the target region is not mapped in the caller's address space.
EPERM	if MC_LOCK, MC_LOCKAS, MC_UNLOCK, or MC_UNLOCKAS is specified and the calling process's effective user ID is not superuser.
EAGAIN	if MC_LOCK or MC_LOCKAS is specified and the limit on primary memory available for locking would be exceeded.
EBUSY	if MC_SYNC with the MS_INVALIDATE option is specified and one or more pages in the target region is locked.
EIO	if MC_SYNC was specified and an I/O error occurred in writing a page.
EIO	if MC_LOCK or MC_UNLOCK was specified and an I/O error occurred in reading a non-resident page.

SEE ALSO

brk(2), exec(2), fork(2), getpagesize(2), mmap(2), mprotect(2), munmap(2), shmat(2), sysconf(2), mlock(3C), mlockall(3C), msync(3C).

NOTES

The DG/UX system's memcntl() implementation does not presently support locking pages of files which have been mapped with the MAP_SHARED attribute.

The DG/UX system's memcntl() implementation presently forces a private copy to be made when a privately mapped page (e.g., a text, data, or stack page) is locked.

Both of these variances from typical System V behavior will be corrected in an upcoming revision of the DG/UX system.

NAME

memctl – set protection of memory mapping

SYNOPSIS

```
#include <sys/m88kbcs.h>

int memctl(void *base, int length, int state);
```

where:

base Starting address of the memory region to be modified

length The length in bytes of the memory region to be modified

state The new memory protection state to which the region should be set

DESCRIPTION

The `memctl()` system call is used to set the state of a region of memory. At any one time, a valid region of memory may be in one of three states:

State	Readable	Writable	Executable
MCT_TEXT	yes	no	yes
MCT_DATA	yes	yes	no
MCT_RONLY	yes	no	no

For COFF format programs, the `exec`(2) functions initialize the text segment's state to `MCT_TEXT`, and initialize the data and stack segments' states to `MCT_DATA`.

Applications should not attempt memory accesses other than those designated above for memory in a given state. The behavior of an improper access attempt to a mapped page is unspecified.

After the last reference (load or store) and before the first execution of any region of shared memory, all processes that have referenced or will execute the region must call `memctl()` to set the state of the given region to `MCT_TEXT`. Similarly, after the last execution and before the first reference of any region of a shared memory, all processes that have executed or will reference the region must issue a `memctl()` call to set the state of the given region to `MCT_DATA` or `MCT_RONLY`; thus, when changing from `MCT_TEXT` the process may set the region to either of the non-executable modes, depending on whether it will write to the shared memory.

The region of memory is defined by the *base* and *length* parameters. The *base* parameter is the starting address of the region, and must be aligned on a memory control unit boundary within the address space. The *length* parameter is the length of the region in bytes, and must be an integral multiple of the memory control unit size. The system's memory control unit size can be obtained using the `sysconf`(2) function. The memory region must be fully mapped within the caller's address space. The *state* parameter must be one of the following values: `MCT_TEXT`, `MCT_DATA`, or `MCT_RONLY`.

ACCESS CONTROL

If the range is part of a shared memory segment which was attached using the `SHM_RDONLY` flag, then *state* must be either `MCT_TEXT` or `MCT_RONLY`.

If any pages in the region have been mapped using `mmap`(2) with the `MAP_SHARED` option, then *state* must be `MCT_TEXT` or `MCT_RONLY` unless the file was open for writing at the time `mmap`(2) was called.

RETURN VALUE

Upon successful completion, `memctl()` returns a value of `0`. Otherwise, it returns the value `-1`, and sets `errno` to indicate an error.

DIAGNOSTICS

Under the following conditions, `memctl(2)` fails and sets `errno` to:

EACCES	if a mapping within the memory region lacks the required permission to change its memory to the requested *state*.
EFAULT	if the memory region is not fully mapped in the caller's address space.
EINVAL	if the *state* parameter is invalid.
EINVAL	if the *base* address parameter is not aligned on a memory control unit boundary.
EINVAL	if the *length* parameter is not an integral multiple of the memory control unit for the system.

SEE ALSO

mmap(2), mprotect(2), shmat(2), sysconf(2).

NAME

mincore – determine residency of memory pages

SYNOPSIS

```
#include &lt;unistd.h>

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

where:

addr Starting address of the memory region to query

len Length in bytes of the memory region to query

vec Pointer to the byte array used to report page status

DESCRIPTION

The `mincore()` function returns the primary memory residency status of pages in the address space covered by mappings in the range [*addr*, *addr* + *len*). The *addr* parameter must be a page aligned address. The *len* parameter is not required to be a multiple of the system page size. However, *len* is rounded up to the next page size multiple before computing the ending address of the range to check. Note that the system page size can be obtained by calling either `getpagesize`(2) or `sysconf`(2) with the `_SC_PAGESIZE` parameter; both calls return identical values.

The status is returned as a byte-per-page in the byte array referenced by *vec*. The byte array must therefore contain one byte for each page in the queried memory region. The least significant bit of each byte is set to `1` to indicate that the referenced page is in primary memory, or to `0` if it is not. The contents of the other bits in each byte are unspecified and reserved for future use; programs should not depend on their values.

The `mincore()` function returns residency information that is accurate at a different instant in time for each page. Because the system may frequently adjust the set of pages in memory, this information may quickly be outdated, and not necessarily even self-consistent. Only locked pages are guaranteed to remain in memory (see `memcntl`(2)).

Pages which are direct mapped to a memory-mapped device will be reported by `mincore()` to be memory resident.

ACCESS CONTROL

No access check is made.

RETURN VALUE

Upon successful completion, `mincore()` returns a value of `0`. Otherwise, it returns the value `-1`, and sets `errno` to indicate the error.

DIAGNOSTICS

Under the following conditions, `mincore()` fails and sets `errno` to:

EINVAL	if the *addr* parameter is not a page aligned address.
ENOMEM	if the *len* parameter is `0`.
ENOMEM	if some portion of the memory region to query is not mapped in the caller's address space.
EFAULT	if some portion of the byte array pointed to by *vec* is not mapped in the caller's address space or lacks write access.

SEE ALSO

`dg_paging_info`(2), `getpagesize`(2), `memcntl`(2), `sysconf`(2).

NAME

mkdir – create a directory file

SYNOPSIS

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

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

where:

path Address of a pathname

mode File mode of the new directory

DESCRIPTION

Path points to a pathname naming a file. If the file does not already exist, a directory of that name is created. The indicated file must be on a file system device mounted read-write. Terminal symbolic links are followed in *path*.

The directory is initialized to contain two entries: '.' and '..', referring to itself and its parent directory. The directory's attributes are set as follows:

- The inode number (st_ino) refers to the per-file database allocated.
- The device number (st_dev) is set to the device code of the containing logical disk unit.
- The represented device (st_rdev) is undefined.
- The number of links (st_nlink) is set to two. (One for the directory's own '.' entry, and one for the entry in the directory's parent.)
- The file size (st_size) is set to reflect the presence of the '.' and '..' entries.
- The file mode (st_mode) is set as follows: The file type is directory. The protection rights are set to the low-order 9 bits of *mode* modified by the process's file mode creation mask; all bits set in the process's creation mask are cleared in the directory's mode (see umask). The set-user-id, set-group-id, and sticky bits are cleared; they have no meaning for a directory.
- The user id (st_uid) is set to the process's effective user id. The group id (st_gid) is set to the process's effective group id.
- The time last accessed (st_atime), time last modified (st_mtime), and time of last attribute change (st_ctime) are set to the current time.

Path is added to the directory's parent and is made to identify the newly created directory. The attributes of the parent directory change as follows:

- The number of links (st_nlink) is incremented, reflecting the '..' entry in the new directory.
- The time last modified (st_mtime) and time of last attribute change (st_ctime) are set to the current time.
- The file size (st_size) is updated.

If the call fails, the directory is not created, and the attributes of the parent remain unchanged.

ACCESS CONTROL

The calling process must have write permission to the parent directory.

The process must have permission to resolve *path*.

RETURN VALUE

0 The new directory was successfully created.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EEXIST	The named file exists.
EIO	An I/O error occurred while writing to the file system device.
EACCES	Permission to add an entry to the parent directory is denied.
EROFS	The named file resides on a file system device mounted read-only.
EMLINK	The maximum number of links to the parent directory would be exceeded by the directory creation.
ENOSPC	No more contiguous space for file space or inodes.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

`mkdir`(1), `rmdir`(1), `rmdir`(2), `stat`(5).

NAME

mknod – create a file entry in the file system

SYNOPSIS

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

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

where:

path	Address of a pathname
mode	Access mode of the new file
device	Device specifier

DESCRIPTION

Path points to a pathname naming a file. Terminal symbolic links are not followed if found in *path*. The file must not exist. The indicated file must be on a file system device mounted read-write.

Device is only pertinent if the file being created is a block or character special file, in which case it is a configuration-dependent specification of a block or character I/O device.

The file's mode (st_mode) is initialized from *mode*. The values of *mode* are constructed by or-ing flags from the following list:

```
file type: (only one may be specified)
        S_IFIFO         0010000         FIFO special.
        S_IFCHR         0020000         Character special.
        S_IFDIR         0040000         Directory.
        S_IFBLK         0060000         Block special.
        S_IFREG         0100000         Ordinary file.

execution mode bits: (any combination)
        S_ISUID         0004000         Set user id.
        S_ISGID         0002000         Set group id.
        S_ISVTX         0001000         Sticky bit.

protection rights: (any combination)
        S_IRUSR         0000400         Read by owner.
        S_IWUSR         0000200         Write by owner.
        S_IXUSR         0000100         Execute by owner.
        S_IRGRP         0000040         Read by group.
        S_IWGRP         0000020         Write by group.
        S_IXGRP         0000010         Execute by group.
        S_IROTH         0000004         Read by other.
        S_IWOTH         0000002         Write by other.
        S_IXOTH         0000001         Execute by other.
```

You cannot make symbolic links , control point directories or socket files via the mknod interface.

If a file type is not specified, it defaults to ordinary. Values of *mode* other than those formed as described above are illegal. *mode* is modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared (see `umask`).

The file's other attributes are initialized as follows:

- The inode number (st_ino) is set to refer to the per-file database allocated.
- The device number (st_dev) is set to the device code containing the logical disk. If the file is block or character special, the represented device (st_rdev) is set to *device*. For other file types, the represented device is undefined.
- The number of links (st_nlink) is set to one.
- The file size (st_size) is set to zero.
- The user id (st_uid) is set to the effective user id of the calling process. The group id (st_gid) is set to the effective group id of the calling process.
- The time last accessed (st_atime), time last modified (st_mtime), and time of last attribute change (st_ctime) are set to the current time.

Path is created in the containing directory and is made to identify the newly created file. The attributes of the directory the file is contained in change as follows:

- The file size (st_size) is updated if the number of blocks necessary to contain all the directory entries increases.
- The time last modified (st_mtime) and time of last attribute change (st_ctime) are set to the current time.

If the call fails, the file is not created, and the attributes of the directory the file is contained in are unchanged.

ACCESS CONTROL

Any process may create a FIFO file, but the effective user id of the process must be superuser to create a directory, special file, or ordinary file.

The process must have write access to the containing directory.

The process must have permission to resolve *path*.

RETURN VALUE

0 The new file was successfully created.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EEXIST	The named file exists.
EINVAL	An invalid file type was specified in *mode*.
EROFS	The directory in which the file is to be created is located on a file system device mounted read-only.
ENOSPC	No more contiguous space left to allocate file space or an inode.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.

ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	Permission to create a directory, special file, or ordinary file is denied.
EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

`mkdir`(1), `chmod`(2), `dg_mstat`(2), `exec`(2), `fstat`(2), `lstat`(2), `stat`(2), `umask`(2), `stat`(5).

NAME

mmap – map pages of memory

SYNOPSIS

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

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

where:

addr	Requested mapping address
len	Length in bytes of the region to map
prot	Protection flags for the new mapping
flags	Mapping control flags
fd	File descriptor of the memory object to be mapped
off	Offset in the file of the region to map

DESCRIPTION

The mmap() function establishes a mapping of a memory object into the caller's address space. The format of the call is as follows:

pa = mmap(*addr*, *len*, *prot*, *flags*, *fd*, *off*);

mmap() establishes a mapping between the process's address space starting at address *pa* for *len* bytes to the memory object represented by the file descriptor *fd* at offset *off* for *len* bytes. The value of *pa* is an implementation-dependent function of the parameter *addr* and values of *flags*, further described below.

The descriptor *fd* may refer to a regular file, as well as to certain block and character special files. Special files usually have device specific restrictions and behavior with respect to mmap(). To avoid repetition of qualifying statements made regarding files, the most typical targets of mmap(), it should be assumed that the following explanations are all applicable to regular files. Exceptions are discussed separately at the end of this section. Also, specific device interactions are discussed in the documentation for each device.

Note that the system page size can be obtained by calling either getpagesize(2) or sysconf(2) with the _SC_PAGESIZE parameter; both calls return identical values.

Three values are used to determine the exact range of file offsets that will be made accessible to user processes by the mmap() function. Two parameters, *len* and *off*, and the size of the file referred to by *fd* are important. (The st_size field of the stat structure is the the precise file size, see stat(2).) The identifier *size* will be used refer to this quantity.

The address range [*pa*, *pa* + *len*) may be accessible to user processes after a successful call to mmap(). There are several cases to consider. First assume that *off* is zero. If *len* is less than or equal to *size*, then all addresses in the address range will be valid. In fact, the entire page containing the last byte in the address range will be valid; this occurs because the system can map areas only in increments of one system page.

If *len* is greater than *size*, then [*pa*, *pa* + *size*), will be valid and accessible, as well as any remainder of the last page in that range. The address range beginning at the next page greater than *pa* + *size*, through the end of the page containing *pa* + *len* – *1*, will be mapped by the mmap() call, but accessing any address in this range will cause the signal `SIGBUS` to be delivered. However, if another process extends the file while it is still mapped by the current process, then any pages in [*pa*, *pa* + *len*) which overlap [*pa* + *old_size*, *pa* + *new_size*) will become accessible by the mapping process. Likewise, any file truncation by another process may result in access faults in [*pa*, *pa* + *len*) if the updated size is made smaller than *len*.

This points out that `mmap()` will return an address range which is "valid" for the user process, but accessing addresses in the range may fail. Note that a valid range does not imply one that can be read and written at all times. The range is valid because it is a reserved part of the process's address space and meets the address space layout requirements. Whether or not particular addresses can be accessed is solely dependent on the structure of the underlying file at the time of the memory access. For example, touching an address may succeed now, but fail later because the file was truncated before the second memory reference.

A special case worth noting occurs in the page which contains the last byte of the mapped file, and when the file size is not an integral multiple of the system page size. Upon the first access to any address in the last page of the file, the entire page will be read from the file into memory, and the region from the last byte of the file to the end of the page will be filled with zeros. Now, as stated above, the calling process may read or modify any data in the entire page. However, when data is stored in the zero filled region beyond the end of the file, it is not guaranteed to be available later. Due to memory demands made by other processes in the system, that page may be written to backing store, but only data up to the current end of file will be written. So, when it is retrieved later, only the data contained in the file will be read, with the rest of the page being zero filled. Hence any data written beyond the end of the file is lost.

Now assume that a non-zero value for the *off* parameter is specified. (Note that the *off* parameter must always be an integral multiple of the system page size.) If *off* + *len* is less than *size*, then [*pa*, *pa* + *len*) is a valid address range, mapping bytes in the file specified by [*off*, *off* + *len*). As above, if *len* is not a multiple of the system page size, then the remainder of the last page in the range will also be mapped. If *off* exceeds *size*, then the entire range [*pa*, *pa* + *len*) will not be accessible to the user process. If *size* is greater than *off*, but *off* + *len* exceeds *size*, and if *SIZE* is shorthand for *size* rounded up the next page aligned value, then [*pa*, *pa* + *SIZE* − *off*) will be accessible, but [*pa* + *SIZE* − *off*, *pa* + *len*) will not. As before, the accessibility of the address range is constant until the size of the mapped file is changed. Recall that other processes may modify the size of the file. Until an address is accessed, it cannot be determined whether accesses in the range will succeed or result in access faults and the delivery of `SIGBUS`.

Finally, note that `mmap()` cannot grow a file. While a successful call may specify *len* larger the the size of the file, only the file system calls, `write`(2) for example, will actually cause the file to expand. No more than *size* bytes will ever be written back to the file.

The parameter *flags* provides other information about the handling of the mapped pages. The options are defined in `<sys/mman.h>` as:

`MAP_SHARED` Share memory modifications.

`MAP_PRIVATE` Memory modifications are private.

`MAP_FIXED` Use *addr* as the mapping start address.

For a successful map, either `MAP_SHARED` or `MAP_PRIVATE` must be specified, but not both.

MAP_SHARED and MAP_PRIVATE describe the disposition of write references to the memory object. If MAP_SHARED is specified, write references will change the mapped file directly. These changes appear immediately; that is, any other process which accesses the file will see the new data. The file object is changed right away, but a delay will occur in writing data to the backing storage. (Note that a system configuration parameter exists to specify the maximum amount of time that may pass before a modified file page is written to its backing storage.)

If MAP_PRIVATE is specified, the initial write reference will create a private copy of the memory object page and redirect the mapping to the copy. No changes to the private copy of the mapped address range will be written to the file referred to by *fd*. Note that the private copy is not created until the first write; until then, other users who have the object mapped MAP_SHARED can change the object, and such changes will be be seen by the process which has the MAP_PRIVATE mapping. Once a private copy of a file page has been made, subsequent modification to the file page will not be reflected in the private copy.

The mapping type is retained across a fork(2).

MAP_FIXED informs the system that the value of *pa* must be *addr*, exactly. When this option is specified, *addr* must be a page aligned address. The use of MAP_FIXED may prevent an implementation from making the most effective use of system resources. Thus, the use of this option is discouraged, except to deliberately replace previous mappings.

When MAP_FIXED is specified, an implicit munmap(2) operation is performed on the address range [*addr*, *addr* + *len*). See munmap(2) for further details.

When MAP_FIXED is not specified, the system uses *addr* in an implementation-defined manner to arrive at *pa*. Currently, the system ignores *addr* completely unless MAP_FIXED is set. The *pa* so chosen will be an area of the address space which the system deems suitable for a mapping of *len* bytes to the specified object. An *addr* value of (caddr_t) 0 grants the system complete freedom in selecting *pa*, subject to the following constraints: address (caddr_t) 0 will never be used, nor will the system replace any extant mapping, nor map into areas considered part of the potential data or stack segments. A value of *addr* other than (caddr_t) 0 is discouraged when MAP_FIXED is not set.

Some implementations add paddings of invalid ranges equivalent to the size of one system page around the mapped region when MAP_FIXED is not specified. So, if *PAGE* is equal to the system page size, and *LEN* is the next page aligned value greater than *len*, then the address ranges [*pa* − *PAGE*, *pa*) and [*pa* + *LEN*, *pa* + *LEN* + *PAGE*) will both be invalid. Such invalid address regions around the mapping are provided to aid in the debugging of user processes with stray memory references. One consequence of this is that use of MAP_FIXED is required to get successive regions mapped at contiguous addresses within a portable application.

When calling mmap() with *fd* referring to a device – a character special or block special file – the *off* and *len* parameters are usually checked during the mmap() call to ensure that the entire range of addresses returned can be accessed at all times. This is unlike the regular file case where *off* can exceed the size of the mapped file and mmap() will complete successfully. Also, certain errno values will be generated only when attempting mmap() on a special file. Specific interactions with special files are described separately in the documentation for the particular device.

The parameter *prot* determines whether read, write, execute, or some combination of accesses are requested for the pages being mapped. Any of these values may be be ORed together, but not all combinations are useful. The protection options are:

PROT_READ The memory can be read.

PROT_WRITE The memory can be written.

PROT_EXEC The memory can be executed.

PROT_NONE The memory cannot be accessed.

Not all implementations literally provide all possible combinations. PROT_WRITE is often implemented as PROT_READ | PROT_WRITE and PROT_EXEC as PROT_READ | PROT_EXEC. However, no implementation will permit a write to succeed where PROT_WRITE has not been set. Also, no implementation will permit any access to succeed where PROT_NONE (alone) has been set.

When implemented on the Motorola 88000 architecture, it is illegal to specify PROT_EXEC and PROT_WRITE together on the same region. Since executable code may be cached separately from data, coherency problems would result if this were allowed. The system will not prevent an attempt to execute writable data, but programs which do so are incorrect and should expect cache coherency problems. Likewise, executing a shared page which another process is writing will produce undefined results. A user process may successfully modify an address range, then use mprotect(2) to change the range's permissions taking away PROT_WRITE and specifying PROT_EXEC.

Note that an mmap() call may fail and leave the user's address space in an unknown state. If mmap() is called with MAP_FIXED, with *addr* and *len* parameters that overlap a previously mapped region, the implicit unmapping may succeed, but the new mapping operation may fail.

ACCESS CONTROL

The file descriptor *fd* must be open with at least read intent.

If MAP_SHARED is specified, the file referenced by *fd* cannot be mapped with PROT_WRITE permission unless the file was opened O_RDWR. Also, any attempt to write into the address range returned by an mmap() call which did not specify PROT_WRITE will cause a SIGSEGV to be delivered, regardless of the type of mapping.

RETURN VALUE

On success, mmap() returns the starting address of the new mapping. On failure it returns (caddr_t) -1 and sets errno to indicate an error.

DIAGNOSTICS

Under the following conditions, mmap() fails and sets errno to:

EINVAL if the *off* parameter is negative.

EINVAL if the quantity *off* + *len* is greater than SIZE_T_MAX.

EINVAL if the *off* parameter is not an integral multiple of the system page size.

EINVAL if the argument *addr* is not a page aligned address and MAP_FIXED is specified.

EINVAL if *addr* + *len* exceeds the largest legal user address and MAP_FIXED is specified.

EINVAL	if the argument *flags* does not contain either MAP_PRIVATE or MAP_SHARED.
EINVAL	if mapping was attempted on *fd* that was not a regular file or device.
EINVAL	if both PROT_WRITE and PROT_EXEC are specified in *prot*.
ENOSYS	if mapping was attempted on a file system which does not support mapping.
ENODEV	if *fd* refers to a device for which mmap() is unsupported.
ENXIO	if [*off*, *off* + *len*) is not a legal range to be mapped as defined by that particular device. This error applies only to character special and block special files.
EIO	if *fd* refers to an NFS file, and record locks are held on the file.
EAGAIN	if *fd* refers to a regular file and mandatory file locking is in effect for the file.
EAGAIN	if the address range could not be locked into memory. This might happen due to a previous call to memcntl(2) which set the MCL_FUTURE option on the process's address space.
EAGAIN	if the mapping uses /dev/zero or MAP_PRIVATE is specified in conjunction with PROT_WRITE, and would reserve more space than the available physical memory and swap space.
EBADF	if *fd* is not a valid, active descriptor.
EACCES	if *fd* is not open for read, regardless of the protection specified, or *fd* is not open for write and PROT_WRITE was specified for a MAP_SHARED mapping.
ENOMEM	if the argument *len* is zero.
ENOMEM	if adding the size of the mapped range would exceed the limit value RLIMIT_AS for the process.
ENOMEM	if MAP_FIXED was specified and the range [*addr*, *addr* + *len*) exceeds that allowed for the address space of a process, or MAP_FIXED was not specified and there is insufficient room in the address space to effect the mapping.

SEE ALSO

fcntl(2), **fork**(2), **getpagesize**(2), **getrlimit**(2), **memcntl**(2), **mprotect**(2), **munmap**(2), **stat**(2), **sysconf**(2).

NOTES

mmap() allows access to resources via address space manipulations instead of the read/write interface. Once a file is mapped, all a process has to do to access it is use the data at the address to which the object was mapped. Consider the following pseudo-code, where *offset* is assumed to be page aligned:

```
fd = open(...)
lseek(fd, offset, SEEK_SET)
read(fd, buf, len)
/* use data in buf */
```

Here is a rewrite using **mmap()**:

```
fd = open(...)
address = mmap((caddr_t) 0, len, (PROT_READ | PROT_WRITE),
                MAP_PRIVATE, fd, offset)
/* use data at address */
```

NAME

`mount` – mount a file system

SYNOPSIS

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

int  mount (const char *special, const char *path, int flag,
                const char fstype, const char *dataptr, int datalen);
```

where:

special	Address of a pathname of a block special file
path	A string indicating the file on which to mount the file system
flag	A bitmask of flags indicating mount options
fstype	The type name for the file system
dataptr	An block address for file system specific data
datalen	The length of the data specified at dataptr

DESCRIPTION

`Mount` adds the file system device identified by *special* to the set of active file system devices, using the file identified by *path* as the mount point. The *flag* contains a bitmask of flags (see below); ordinarily the MS_DATA flag must be set. The *dataptr* and *datalen* describe the block address and the length of file system specific data. Mount has the following consequences:

- The filename store contained on *special* is added to the system filename store. Thus, all files contained on *special* can be named.
- References to the mount point will refer to the root directory on the mounted file system device.
- The original sub-tree under the mount point disappears from the system filename store. However, the files in that subtree remain unchanged. These files still exist, but can no longer be named. Already opened file descriptors for these files will remain valid.

Flag contains the following bitmap options, defined in <sys/mount.h>:

MS_DATA
: This is ordinarily required; it indicates the arguments *fstype*, *dataptr*, and *datalen* are being used. (For backward compatibility, if this flag is not set, then fstype is assumed to be the same as the root file system, and *dataptr* and *datalen* assumed to be zero.)

MS_RDONLY
: If this is set, then any writing to the file system is not allowed. Otherwise writing is controlled by individual file permissions.

MS_NOSUID
: This indicates the file system does not support setuid and setgid semantics.

MS_REMOUNT
: This flag indicates the file system is already mounted and any associated attributes of the mount should be modified to that of this call. This is used to change options, however not all changes are possible. For example, it t is impossible to make a currently mounted writeable file system to be read only.

If an error occurs, no changes are made.

ACCESS CONTROL

The effective user id of the calling process must be superuser. The exception to this is when the string *namefs* is use for the value of *fstype*. In this case, the effective user id of the calling process must be superuser, or the effective user id must be the owner of *path* and have write permission to *path*.

RETURN VALUE

0	Completed successfully.
−1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBUSY	*Path* is being used by another mount, is someone's current working path or is otherwise open for access.
EBUSY	The device associated with *special* is currently mounted.
EBUSY	The system limit on mounted devices has been reached.
EINVAL	System information on the file system is bad.
ENOSPC	Not enough memory was available to read system information from the file system.
EIO	An I/O error occurred while reading system information from the file system.
ENOTBLK	*Special* is not a block special device.
ENOTDIR	*Path* is not a directory and the file system type requires a directory.
ENXIO	The device associated with *special* does not exist.
EPERM	Permission to mount a file system device is denied to the calling process.
EROFS	*Path* resides on a read-only file system.
ENOENT	Either *special* or *path* do not exist.
ENOENT	A non-terminal component of either *special* or path does not exist.
ENOTDIR	A non-terminal component of either *special* or *path* was not a path or symbolic link.
ENAMETOOLONG	Either *special* or *path* exceeds the length limit for pathnames.
ENAMETOOLONG	A component of either *special* or *path* exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve either *special* or *path* or to expand a symbolic link.
ELOOP	The number of symbolic links encountered while resolving either *special* or *path* exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	Either *special* or *path* contains a character not in the allowed character set.
EFAULT	Either *special* or *path* does not completely reside in the process's address space or either *special* or *path* does not

terminate in the process's address space.

SEE ALSO

dg_mount(2), umount(2). fattach(3).

NAME

mprotect – set protection of memory mapping

SYNOPSIS

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

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

where:

addr Starting address of the memory region to modify

len Length in bytes of the memory region to modify

prot New protections to apply to the memory region

DESCRIPTION

The `mprotect()` function changes the access protections on the mappings specified by the range [*addr*, *addr* + *len*) to be those specified by *prot*. The *len* parameter is rounded up to the next page size multiple before computing the ending address of the range upon which to set protections. Note that the entire memory region must be mapped within the caller's address space.

The *addr* parameter must be a page aligned address. The system page size can be obtained by calling either `getpagesize`(2) or `sysconf`(2) with the `_SC_PAGESIZE` parameter; both calls return identical values.

The parameter *prot* determines whether read, write, execute, or some combination of accesses are requested for the pages being modified. Any of these values may be be ORed together, but not all combinations are useful. The protection options are:

`PROT_READ` The memory can be read.

`PROT_WRITE` The memory can be written.

`PROT_EXEC` The memory can be executed.

`PROT_NONE` The memory cannot be accessed.

Not all implementations literally provide all possible combinations. `PROT_WRITE` is often implemented as `PROT_READ | PROT_WRITE` and `PROT_EXEC` as `PROT_READ | PROT_EXEC`. However, no implementation will permit a write to succeed where `PROT_WRITE` has not been set. Also, no implementation will permit any access to succeed where `PROT_NONE` (alone) has been set.

When implemented on the Motorola 88000 architecture, it is illegal to specify `PROT_EXEC` and `PROT_WRITE` together on the same region. Since executable code may be cached separately from data, coherency problems would result if this was allowed. The system will not prevent an attempt to execute writable data, but programs which do so are incorrect and should expect cache coherency problems. Likewise, executing a shared page which another process is writing will produce undefined results. A user process may successfully modify an address range, then use `mprotect`(2) to change the range's permissions taking away `PROT_WRITE` and specifying `PROT_EXEC`.

If the `mprotect()` function fails, portions of the specified address range may have had their protections changed, while others have not.

ACCESS CONTROL

If the range is part of an `mmap`(2) region, and it has been mapped with the `MAP_SHARED` option, then `PROT_WRITE` cannot be specified in *prot* unless the file was open for writing at the time `mmap`(2) was called.

If the range is part of a shared memory segment which was attached using the `SHM_RDONLY` flag, then `PROT_WRITE` cannot be specified in *prot*.

RETURN VALUE

Upon successful completion, `mprotect()` returns a value of `0`. Otherwise, it returns the value `-1`, and sets `errno` to indicate an error.

DIAGNOSTICS

Under the following conditions, `mprotect()` fails and sets `errno` to:

`EINVAL` if *addr* is not a page aligned address.

`EINVAL` if both `PROT_WRITE` and `PROT_EXEC` are specified in *prot*.

`EACCES` if *prot* specifies a protection that violates the access permissions some mapping in the region has to its underlying memory object.

`EAGAIN` if *prot* specifies `PROT_WRITE` over a locked `MAP_PRIVATE` mapping and there are insufficient memory resources to reserve for locking the private page.

`EAGAIN` if *prot* specifies `PROT_WRITE` over a `MAP_PRIVATE` mapping and there are insufficient swap space resources to be reserved for possible page modification.

`ENOMEM` if *len* is equal to zero.

`ENOMEM` if some page in the memory region is not mapped within the caller's address space.

SEE ALSO

`getpagesize`(2), `memctl`(2), `mmap`(2), `shmat`(2), `sysconf`(2).

NAME

`msgctl` – get or set message queue attributes or destroy a message queue

SYNOPSIS

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

int  msgctl (msqid, cmd, buf)
int  msqid;
int  cmd;
struct msqid_ds * buf;
```

where:

msqid — A message queue identifier

cmd — The message control operation to be performed

buf — The address of a message queue attribute record (used only if *cmd* is IPC_STAT or IPC_SET)

DESCRIPTION

`Msgctl` is used to get and set message queue attributes or to destroy a message queue. The subject message queue is identified by *msqid*. The action performed by `msgctl` depends on the value of *cmd* as follows:

IPC_STAT — The user-visible msqid_ds structure is returned in *buf*. If an error occurs, the contents of *buf* are undefined.

IPC_SET — The following message queue attributes are set to the values found in the structure pointed to by *buf*: user id (msg_perm.uid), group id (msg_perm.gid), permission rights (in msg_perm.mode), and the maximum size (msg_qbytes).

If an error occurs, the message queue remains unchanged. Otherwise, the last change time (msg_ctime) is set to the current time.

IPC_RMID — The message queue is destroyed. All resources consumed by the message queue are freed and the message queue identifier is invalidated. All queued messages are lost.

If an error occurs, the message queue remains unchanged.

ACCESS CONTROL

Operation permission depends on the value of *cmd* as follows:

IPC_STAT — The calling process is required to have read access to the message queue.

IPC_SET — The effective user id of the calling process must be equal to the message queue's user id, the message queue creator's user id, or that of the superuser. If the maximum size of the message queue is increased, the effective user id of the calling process must be the superuser.

IPC_RMID — The effective user id of the calling process must be equal to the message queue's user id, the message queue creator's user id, or that of the superuser.

RETURN VALUE

0 Completed successfully.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes regardless of the value of *cmd*:

EINVAL *msqid* is not a valid message queue identifier.

EINVAL *cmd* is not a valid command.

If *cmd* is IPC_STAT, `errno` may be set to one of these values:

EACCES Read permission is denied to the calling process.

EFAULT *buf* points to an illegal address.

If *cmd* is IPC_SET, `errno` may be set to one of these values:

EPERM Permission to change the message queue attributes is denied to the calling process.

EPERM Permission to increase to the maximum size of the message queue is denied to the calling process.

EFAULT *buf* points to an illegal address.

If *cmd* is IPC_RMID, `errno` may be set to this value:

EPERM Permission to remove the message queue is denied to the calling process.

SEE ALSO

`intro`(2), `ipcrm`(1), `ipcs`(1), `msgget`(2), `msgrcv`(2), `msgsnd`(2).

NAME

msgget – get message queue identifier

SYNOPSIS

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

int    msgget (key, msgflg)
key_t  key;
int    msgflg;
```

where:

key A user-defined name for the message queue

msgflg A set of flags indicating the requested permission state of the message queue, whether a new message queue should be created, and whether the message queue should be held exclusively

DESCRIPTION

Msgget returns the message queue identifier associated with *key*.

This message queue identifier may then be used in other message queue operations as specified by msgctl, msgsnd, and msgrcv.

Msgget can be used to get the message queue identifier of an existing message queue or to create a new message queue as follows:

Four options are available:

- Create a private message queue.

  In this case, *key* is IPC_PRIVATE.

  A process can create a "private" message queue by using the special IPC_PRIVATE key. The system will create a message queue identifier that is private to the process. The message queue identifier will not be returned to other processes regardless of what key value they specify.

  The newly created message queue can be shared among other processes by distributing the message queue identifier.

  A process can make multiple msgget operations specifying IPC_PRIVATE. The identifiers returned will be unique and the associated message queues will be different.

- Find *key* if already defined.

  In this case, the IPC_CREAT and IPC_EXCL bits of *msgflg* are clear and *key* is not IPC_PRIVATE.

  The message queue identifier associated with the given key is returned. If none exists or if one exists but the permission rights of the message queue do not include those specified by the low-order 9 bits of *msgflg*, an error is returned.

- Create only if *key* is not already defined.

  In this case, the IPC_CREAT and IPC_EXCL bits of *msgflg* are both set and *key* is not IPC_PRIVATE.

If a message queue identifier already exists for *key* an error is returned. Otherwise, a message queue identifier and associated message queue are created. The message queue identifier will be returned to other processes that specify the same key value.

- Find *key* if already defined, otherwise create.

  In this case, the IPC_CREAT bit of *msgflg* is set, the IPC_EXCL bit of *msgflg* is clear, and *key* is not IPC_PRIVATE.

  If a message queue identifier already exists for *key*, this is identical to the second option above. Otherwise, this is identical to the third option above.

If a new message queue is created, its attributes are initialized as follows:

- The message queue creator's user id (msg_perm.cuid) and the message queue's user id (msg_perm.uid) are set to the effective user id of the calling process.
- The message queue creator's group id (msg_perm.cgid) and the message queue's group id (msg_perm.gid) are set to the effective group id of the calling process.
- The message queue's permission rights (in msg_perm.mode) are set to the low-order 9 bits of *msgflg*.
- The current size (msg_cbytes), the process id performing the last `msgsnd` and `msgrcv` operations (msg_lspid and msg_lrpid), and the times of the last `msgsnd` and `msgrcv` operations (msg_stime and msg_rtime) are all set to their initial values.
- The most recent time the message queue attributes were changed (msg_ctime) is set to the current time.
- The maximum size (msg_qbytes) is set to the system limit.

ACCESS CONTROL

See the description of the exception condition EACCES below.

RETURN VALUE

msqid A non-negative integer that identifies the message queue associated with *key*.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

If a message queue exists for *key*, `errno` may be set to one of these values:

EACCES The permission rights of the message queue do not include those specified by the low-order 9 bits of *msgflg*.

EEXIST Both the IPC_CREAT and IPC_EXCL bits of *msgflg* are set.

If a message queue does not exist for *key*, `errno` may be set to one of these values:

ENOENT The IPC_CREAT bit of *msgflg* is not set.

ENOSPC Creating a new message queue would cause the system-imposed limit on the number of message queues to be exceeded.

SEE ALSO

intro(2), ipcrm(1), ipcs(1), msgctl(2), msgrcv(2), msgsnd(2).

NAME

msgrcv – receive a message

SYNOPSIS

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

int    msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int    msqid;
struct msgbuf * msgp;
size_t msgsz;
long   msgtyp;
int    msgflg;
```

where:

msqid A message queue identifier

msgp A buffer for the message

msgsz The size in bytes of the message to be received

msgtyp Message type

msgflg A set of flags qualifying the action of msgrcv

DESCRIPTION

Msgrcv reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the message buffer pointed to by *msgp*.

Msgtyp is used to select from other queued messages; *msgsz* bytes of such a message (only a single message, if any, is selected in a call to this routine) are returned.

If the received message is larger than *msgsz* and the MSG_NOERROR bit of *msgflg* is set, the received message is truncated to *msgsz* bytes. The truncated part of the message is lost and no indication of the truncation is given to the calling process. If the received message is larger than *msgsz* and the MSG_NOERROR bit of *msgflg* is clear, an error is returned.

Msgtyp specifies the type of message requested as follows:

- *msgtyp* == 0: The first message on the queue is received.
- *msgtyp* > 0: The first message of type *msgtyp* is received.
- *msgtyp* < 0: The first message of the lowest type of all messages on the queue is received provided the type is less than or equal to the absolute value of *msgtyp*.

Msgflg specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

- If the IPC_NOWAIT bit of *msgflg* is set, the calling process will return immediately.
- If the IPC_NOWAIT bit of *msgflg* is clear, the calling process will be suspended until:
- A message of the desired type is placed on the queue, in which case, the operation is successful,
- *msqid* is removed from the system, in which case, msgrcv will return with the error condition EIDRM, or

- The calling process receives a signal that is to be caught, in which case, msgrcv will return with the error condition EINTR.

If msgrcv fails, the message queue will be unchanged. Upon successful completion, the message queue attributes are changed as follows:

- The number of messages on the queue (msg_qnum) is decremented.
- The number of bytes on the queue (msg_cbytes) is reduced by the size of the mtext portion of the received message.
- The process id of the last process performing a msgrcv operation (msg_lrpid) is set to that of the calling process.
- The most recent time a msgrcv operation was performed (msg_rtime) is set to the current time.

ACCESS CONTROL

Read access to the message queue is required.

RETURN VALUE

actual_size	Completed successfully. The number of bytes actually placed into mtext.
−1	An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EINVAL	*msqid* is not a valid message queue identifier.
EACCES	Read permission is denied to the calling process.
EINVAL	*msgsz* is less than 0.
E2BIG	*msgp*-mtext> is greater than *msgsz* and the MSG_NOERROR bit of *msgflg* is not set.
ENOMSG	The queue does not contain a message of the desired type and the IPC_NOWAIT bit of *msgflg* is set.
EFAULT	*msgp* points to an illegal address.
EIDRM	*msqid* was removed from the system while the calling process was suspended by msgrcv.
EINTR	The calling process received a signal that was set to be caught while suspended by msgrcv.

SEE ALSO

intro(2), ipcrm(1), ipcs(1), msgctl(2), msgget(2), msgsnd(2), signal(2).

NAME

msgsnd – send a message

SYNOPSIS

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

int     msgsnd (msqid, msgp, msgsz, msgflg)
int     msqid;
struct msgbuf *msgp;
size_t msgsz;
int     msgflg;
```

where:

msqid A message queue identifier

msgp The message buffer of the message to be sent

msgsz The size in bytes of the *mtext* portion of the message buffer

msgflg A set of flags modifying the action of msgsnd

DESCRIPTION

Msgsnd sends a message to the queue associated with the message queue identifier specified by *msqid*. *msgp* points to the user's message buffer containing a message type used for message selection, and the text of the message. *msgsz* is the length of the message. It can range from 0 to a configurable system-imposed maximum.

msgflg specifies the action to be taken if either the number of bytes already on the queue after this message is added would exceed the maximum queue size, or the total number of messages on all queues system-wide is equal to the system-imposed limit. If either of these conditions hold, the following actions are taken:

- If the IPC_NOWAIT bit of *msgflg* is set, the message will not be sent and the calling process will return immediately.
- If the IPC_NOWAIT bit of *msgflg* is clear, the calling process will be suspended until:
  - the condition responsible for the suspension no longer exists, in which case, the operation is successful,
  - *msqid* is removed from the system, in which case, msgsnd will return with the error condition EIDRM, or
  - the calling process receives a signal that is to be caught, in which case, msgsnd will return with the error condition EINTR.

If msgsnd fails, no message is sent and the message queue is unchanged. Upon successful completion, the message queue attributes are changed as follows:

- The number of messages on the queue (msg_qnum) is incremented.
- The number of bytes on the queue (msg_cbytes) is increased by the size of the *mtext* portion of the message being sent.
- The process id of the last process performing a msgsnd operation (msg_lspid) is set to the calling process.
- The most recent time a msgsnd operation was performed (msg_stime) is set to the current time.

ACCESS CONTROL

Write access to the message queue is required.

RETURN VALUE

0 Completed successfully.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EINVAL *msqid* is not a valid message queue identifier.

EINVAL Message type is less than 1.

EINVAL *msgsz* is less than zero or greater than the system-imposed limit.

EACCES Write permission is denied to the calling process.

EAGAIN The message cannot be sent for some reason and the IPC_NOWAIT bit of *msgflg* is set.

EFAULT *msgp* is an illegal address.

EIDRM *msqid* was removed from the system while the calling process was suspended by `msgsnd`.

EINTR The calling process received a signal that was set to be caught while suspended by `msgsnd`.

SEE ALSO

`intro`(2), `ipcrm`(1), `ipcs`(1), `msgctl`(2), `msgget`(2), `msgrcv`(2), `signal`(2).

NAME

msgsys – perform a message queue operation

SYNOPSIS

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

int   msgsys (P1, P2, P3, P4, P5, P6)
int   P1;
int   P2;
int   P3;
int   P4;
int   P5;
int   P6;
```

where:

P1 An argument indicating the type of operation to be performed with message queues (0 = MSGGET, 1 = MSGCTL, 2 = MSGRCV, 3 = MSGSND).

P2 If the operation is MSGGET, *P2* is equal to the message queue key. Otherwise, *P2* is equal to the message queue id.

P3 If the operation is MSGGET, *P3* equals the flags that indicate whether to create the queue or not and the permissions for the queue. If the operation is MSGCTL, *P3* equals the control command number which specifies the type of control command to perform. If the operation is MSGSND or MSGRCV, *P3* equals the pointer to the message to be sent or received.

P4 If the operation is MSGGET, *P4* is invalid. In case of MSGCTL, *P4* is a pointer to a buffer containing information about the message queue. In case of MSGSND and MSGRCV, *P4* is equal to the size of the message's text portion.

P5 If the operation is MSGGET or MSGCTL, *P5* is invalid. In case of MSGRCV *P5* is equal to the message type. In case of MSGSND *P5* is equal to the message flags modifying the message MSGSND operation.

P6 If the operation is MSGRCV, *P6* is equal to the message flags modifying the MSGRCV operation. Otherwise *P6* is invalid.

DESCRIPTION

`Msgsys`(2) performs a message operation (MSGGET, MSGCTL, MSGSND, MSGRCV) indicated by the value of *P1*.

ACCESS CONTROL

See the description of the exception condition EACCES below.

RETURN VALUE

msqid A non-negative integer that identifies the message queue associated with *key*, returned by `msgget`.

0 `Msgsnd`, `msgrcv` or `msgctl` calls were successful.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

The error codes returned depend on the type of message queue operations performed and are described in `msgget`(2), `msgctl`(2), `msgsnd`(2), `msgrcv`(2).

EINVAL *P1* argument is not in the range of 0 through 3.

SEE ALSO

intro(2), msgctl(2), msgget(2), msgrcv(2), msgsnd(2).

NAME

munmap – unmap pages of memory

SYNOPSIS

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

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

where:

addr Starting address of the memory region to unmap
len Length in bytes of the memory region to unmap

DESCRIPTION

The munmap() function removes any mappings for pages in the range [*addr*, *addr* + *len*). Such mappings in the address range will be removed regardless of how they were established. References to unmapped pages will result in the delivery of a SIGSEGV signal to the process. Note that the mmap(2) function performs an implicit munmap() operation when MAP_FIXED is specified in the *flags* parameter.

The *addr* parameter must specify a page aligned address. The system page size is available by calling either getpagesize(2) or sysconf(2) with the _SC_PAGESIZE parameter; both calls return identical values.

The *len* parameter is rounded up to the next multiple of the page size before computing the ending address of the range to unmap.

The *addr* and *len* parameters to munmap() operations do not have to match the similar parameters to mmap(2) calls which may have established the mapping. Munmap() can release the address range of any part of a mapped region, at the beginning, ending, or middle of it. Also, the [*addr*, *addr* + *len*) interval may span regions of multiple mmap(2) calls, possibly containing addresses not currently mapped in the caller's address space. Further, not only mmap(2) regions can be unmapped by munmap(). Any part of a process's address space which overlaps the range [*addr*, *addr* + *len*) will be unmapped, even if the segment is part of the data segment, stack, or a shared memory segment.

When the given address range spans a shared memory region, the attached shared memory segment will be unmapped. However, such segments will still be considered attached by the system, which may cause unexpected results. Processes should avoid the use of munmap() on ranges of shared memory addresses, and instead use shmdt(2) to cleanly detach the shared memory segment.

Regions of the process's data segment (allocated via brk(2), sbrk(2), and malloc(3C)) can also be invalidated with munmap(). This practice is not recommended since it does not alter the caller's break value, and thus may lead to unexpected results.

Since the [*addr*, *addr* + *len*) range may span unmapped portions of the caller's address space, the actual memory unmapped by munmap() may be smaller than *len*. The size of the caller's virtual address space as accounted for by the system will decrease only by the sum of the ranges of newly unmapped pages.

Note that any pages within the range which were locked into memory via memcntl(2) or any of its associated library routines will be unlocked when unmapped.

ACCESS CONTROL

No access check is made.

RETURN VALUE

Upon successful completion, `munmap()` returns a value of `0`. Otherwise, it returns the value `-1`, and sets `errno` to indicate an error.

DIAGNOSTICS

Under the following conditions, `munmap()` fails and sets `errno` to:

`EINVAL`	if *len* is equal to zero.
`EINVAL`	if *addr* is not a page aligned address.
`EINVAL`	if *addr+len* exceeds the largest legal user address.

SEE ALSO

`memcntl`(2), `mmap`(2), `getpagesize`(2), `sysconf`(2).

NAME

`nfssvc` – start an NFS server on a specified socket

SYNOPSIS

```
int  nfssvc (socket)
int  socket;
```

where:

socket Socket to listen to requests on

DESCRIPTION

An NFS server (daemon) is started on the socket identified by *socket*. *socket* is the descriptor obtained from a `socket(2)` system call. The socket must be AF_INET, and SOCK_DGRAM (protocol UDP/IP). This system call does not normally return.

ACCESS CONTROL

None.

RETURN VALUE

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF The descriptor indentified by *socket* is out of range, or the descriptor is not active.

EINVAL The socket indentified by *socket* does not specify a socket object.

EINTR The process was terminated by a signal.

EOPNOTSUPP Kernel support for NFS is not present.

SEE ALSO

`socket(2)`.

NAME

`nice` – change priority of a process

SYNOPSIS

```
#include <unistd.h>

int  nice (incr)
int  incr;
```

where:

incr A positive or negative value that is to be added to the calling process's priority

DESCRIPTION

The value of *incr* is added to the priority of the calling process. A more positive priority value results in a lower level of service from the CPU.

If the new priority would be greater than 19, the process's priority is set to 19. If the new priority would be less than −20, the process's priority is set to −20.

ACCESS CONTROL

The effective-user-id of the calling process must be 0 (super-user) for nice to accept a value for *incr* that is less than 0 or greater than 39. If this condition is not met, *incr* will be treated as 0, and `errno` will be set to EPERM.

RETURN VALUE

`Nice` always returns the calling process's priority upon completion of the system call. If an error occurred on the call, the process's priority will be unchanged and `errno` will be set to indicate the error. `errno` should be set to zero before the call and checked afterwards, regardless of the return value.

DIAGNOSTICS

EPERM The value of *incr* is negative or greater than 39 and the effective-user-id of the calling process is not 0.

SEE ALSO

`exec(2)`.

NAME

open – open file for reading or writing

SYNOPSIS

```
#include <fcntl.h>

int  open (path, open_flag, protection_mode)
char * path;
int  open_flag;
int  protection_mode;
```

where:

path	Address of a pathname
open_flag	Open intent and open behavior flags
protection_mode	Protection mode, if file is created

DESCRIPTION

Path points to a pathname naming a file to be opened. Terminal symbolic links are followed in *path*. *open_flag* is a group of flags specifying the open intent (read, write, or both) and requests for optional behavior of the call. It is constructed by or-ing the desired flags. One and only one of the following three open intents must be specified in *open_flag*:

- O_RDONLY 0
- O_WRONLY 1
- O_RDWR 2

Ignoring for the moment all flags except the open intents, if the file exists, the semantics of an open are:

- Ordinary, FIFO, block special, and character special files may be opened for any of the intents. Directories can only be opened for O_RDONLY intent.
- If the specified intent is O_RDWR or O_WRONLY and the file's type is ordinary or FIFO, the file must reside on a file system device mounted read-write.
- The lowest numbered available file descriptor is allocated, and the file pointer is set to the beginning of the file.
- If the file's type is block or character special, a device driver is called to perform device dependent initialization.

If the file does not exist, and the O_CREAT flag has not been specified, then the call fails.

The basic semantics of the open call described above may be modified by setting one or more of the following flags in *open_flag*:

O_NDELAY This flag has differing semantics depending on the type of file or or device it is referencing. If the file is a FIFO and O_NDELAY is set, a reader of a FIFO file does not pend during the open, waiting for the presence of a writer. A writer of such a FIFO file does not pend, either, but the error ENXIO is asserted if no reader is present.

If the file is a FIFO and O_NDELAY is not set, then a reader of the file will pend waiting for a writer of the file to open the FIFO, and likewise, a writer will pend waiting for a reader to open the FIFO.

A process opening the FIFO for both read and write is not affected by this flag.

If the file is associated with a communications device, and O_NDELAY is set, then an opener for any intent will not wait for a carrier to be present on the line before returning from the `open` call.

If the file is associated with a communications device, and O_NDELAY is not set, then an `open` will pend waiting for a carrier to be present on the line.

This flag is "remembered" in the object pointer's flags and affects subsequent reads and writes. See `read`(2) and `write`(2).

O_CREAT
If set, the O_CREAT flag guarantees that the file exists after the open call is completed. If it is set and the file already exists, the open occurs as described above. If the file does not exist, an ordinary file with the name *path* is created and then the file is opened for the intent requested. The file must be on a file system device mounted read-write. It is created in the manner of the `creat` system call:

The file is entered into the flat file store by allocating and initializing a per-file database. The file's attributes are set as follows:

- The inode number (st_ino) refers to the per-file database allocated.
- The file's device (st_dev) is set to the device code of the logical disk unit that contains the new file.
- The represented device (st_rdev) is undefined.
- The file size (st_size) is set to 0.
- The number of links (st_nlink) is set to one.
- The user id (st_uid) is set to the effective user id of the calling process. The group id (st_gid) is set to the process's effective group id.
- The file mode (st_mode) is set as follows: The file type is ordinary. The sticky bit is cleared. The protection rights, set-user-id, and set-group-id bits of the file mode are set to the value of *protection_mode* modified by the process's file mode creation mask; all bits set in the mask are cleared in the file mode (see `umask`). The set-group-id bit is set only if the file's group id is the same as the process's effective group id or is in the process's group set.
- The time last accessed (st_atime), time last modified (st_mtime), and time of last attribute change (st_ctime) are set to the current time.

- *Path* is added to the filename store (i.e., a link is created in the containing directory) and is made to identify the newly created file. An allocation to the directory causes its attributes to change as follows:
- The file size (st_size) may be updated.
- The time last modified (st_mtime) and time of last attribute change (st_ctime) are set to the current time.

O_EXCL
The O_EXCL flag modifies the O_CREAT flag and has no effect if O_CREAT is clear or the file does not exist. If O_CREAT and O_EXCL are set, the open will fail if the file already exists. The O_EXCL flag also interacts with symbolic links in the following way. If O_EXCL is on (with O_CREAT), and the last component of the path is a symlink, then the open will fail even if the symlink points to a non-existent file.

O_TRUNC
This flag implies that you are opening the file for write intent, even though the user may have specified a read-only channel to be opened. Thus, a channel created with this flag on is always open for write intent.

O_TRUNC has no effect if the file does not exist. File specific ramifications of this flag are:

- Directories cannot be truncated. You can never gain a write-accessable channel to a directory, so you can never truncate them through this interface.
- Ordinary and FIFO files being truncated must reside on a file system device mounted read-write.
- If the file's type is ordinary, the file's disk blocks are freed and its size (st_size) is set to zero.
- The file's time last modified (st_mtime) and time of last change to the attributes (st_ctime) are set to the current time. (This happens whether the file's contents were changed or not.)
- All other file attributes remain unchanged.

O_APPEND
The O_APPEND flag has no visible effect on the operation of the open call. If set, it is "remembered" as part of the file's open intents and will affect subsequent writes by positioning the file pointer to the end of the file prior to each write.

O_SYNC
The O_SYNC flag has no visible effect on the operation of the open call. If set, it is "remembered" as part of the file's open intents and will affect subsequent writes by forcing all changes to the file to disk before returning from the write call. File changes include changes to any data buffers and inode information.

O_NONBLOCK
This flag has differing semantics depending on the type of file or device it is referencing.

If the file is a FIFO and O_NONBLOCK is set, a reader of a FIFO file does not pend during the open, waiting for the presence of a writer. A writer of such a FIFO file does not pend, either, but the error ENXIO is asserted if no reader is present.

If the file is a FIFO and O_NONBLOCK is not set, then a reader of the file will pend waiting for a writer of the file to open the FIFO, and likewise, a writer will pend waiting for a reader to open the FIFO.

A process opening the FIFO for both read and write is not affected by this flag.

If the file is a block or character special file and O_NONBLOCK is set, then an opener for any intent will not wait for a device to be ready or available before returning from the `open` call. Subsequent behavior of the device is device specific.

If the file is a block or character special file and O_NONBLOCK is not set, then an `open` will pend waiting for the device to be ready or available.

O_DG_UNBUFFERED
: Normally, the default behavior for acquiring data from an ordinary file is to use the system buffer cache to cache requests for the data from the file and then to copy data from the system buffer into the user's buffer. The presence of this flag will change the default behavior and access method for acquiring data from the file. Specifically, `read`(2) and `write`(2) will not operate, but the system calls, `dg_unbuffered_read`(2) and `dg_unbuffered_write`(2) will work. `dg_unbuffered_read`(2) and `dg_unbuffered_write`(2) transfers blocks of file data from the disk directly to or from the user's buffer in a synchronous manner. Upon successful opening of the file with this flag, the buffer cache for the file will have been flushed to disk and invalidated. Any attempts to use `read`(2) or `write`(2) on the descriptor will return an error. Descriptors returned with this flag differ in no other way from other descriptors returned without this flag being set. This call will fail if there are other descriptors for the file that were opened without this flag set. Also, `open` calls without this flag will fail if there are descriptors to the file that have the flag set. This flag cannot be set or unset via the `fcntl`(2) interface.

O_DG_SHARED_DESCRIPTOR
: By default, descriptors are part of the per-process data of the process that creates them. The use of this flag in the `open` call will change this behavior. If set, the descriptor created by the open will exist in the shared descriptor table for the process and be accesible to all processes that have attached the shared descriptor array via `dg_attach_to_shared_descriptors`(2). Descriptors in this shared table have different reference count semantics from normal descriptors. See the manual page for `dg_attach_to_shared_descriptors`(2) for details.

Bits in *open_flag* other than those flags mentioned above are undefined and should not be used.

The *mode* parameter is used only when the file is created, i.e., when O_CREAT is set and the file does not already exist. In other cases, it is ignored.

Note that `creat(`*path*`,` *mode*`)` has the same semantics as `open(`*path*`,O_WRONLY|O_CREAT|O_TRUNC,`*mode*`)` – If the file exists, it is truncated; if it does not exist, it is created; in both cases it is opened for writing.

If the process exceeds its limit on open files, the `open` call will fail, and the file will be left in the state it was in before the call. The limit on per-process descriptors is bounded above by the soft limit on per-process descriptors for the process. A process may raise this soft limit by calling `setrlimit`(2). The current soft limit may be found by calling `getrlimit`(2). The soft limit may only be raised until the system wide hard limit is reached.

Upon successful completion, the descriptor is returned. The descriptor is set to remain open across `exec` calls. See `fcntl`(2).

ACCESS CONTROL

To open an existing file, the calling process must have read and/or write access (as requested) to the file.

To create a file, the process must have write access to the containing directory.

To truncate an existing file, the process must have write access to the file.

The process must have permission to resolve *path*.

RETURN VALUE

Any non-negative integer
: The file descriptor for the successfully opened file.

–1
: An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EACCES	The open intents specified in *open_flag* are denied for the named file or if in creating the file, the target containing directory disallows access.
EINVAL	Invalid argument passed to this function.
EEXIST	O_CREAT and O_EXCL are set, and the named file exists, or is pointed at by a symbolic link.
EINTR	A signal was caught during the open system call.
EISDIR	The named file is a directory and the open intent is write or read/write.
EMFILE	NOFILE file descriptors are currently open. You have reached the soft limit on file descriptors. If you wish to open another file, then you must increase the number of available descriptors with the `getrusage`(2) and `setrusage`(2) system calls.
ENOENT	O_CREAT is clear and the named file does not exist; or the file the pathname resolved to does not exist and O_CREAT was not specified; or a non-terminal component of the pathname does not exist.
ENXIO	The named file is a character special or block special file, and the device associated with this special file does not exist; or O_NDELAY or O_NONBLOCK is set, the named file is a

	FIFO file, O_WRONLY is set, and no process has the file open for reading.
EOPNOTSUPP	An attempt was made to open a socket.
EROFS	The named file resides on a file system device mounted read-only and the open intent is write or read/write.
ENOSPC	No more contiguous space to create a file entry or inode.
EAGAIN	File exists with record locks in mandatory enforcement mode and O_CREAT and/or O_TRUNC is specified.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames; or a component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.
ENOSR	The *path* is STREAMS-based and the system is unable to allocate a stream.
EIO	if during the `open()` of a STREAMS-based device, a hangup or error occurs.

SEE ALSO

`chmod`(2), `close`(2), `creat`(2), `dup`(2), `fcntl`(2), `lseek`(2), `read`(2), `umask`(2), `write`(2), `fcntl`(5), `stat`(5), `dg_allow_shared_descriptor_attach`(2), `dg_attach_to_shared_descriptors`(2).

NAME

pathconf, fpathconf – get configurable pathname variables

SYNOPSIS

```
#include <unistd.h>

long pathconf (path, name)
char *path;
int name;

long fpathconf (fildes, name)
int fildes, name;
```

where:

path The name of a pointer to the pathname of a file or directory
name The variable to be queried relative to the file or directory
filedes An open file descriptor

DESCRIPTION

The pathconf() and fpathconf() functions provide a method for the application to determine the current value of a configurable limit or option (*variable*) that is associated with a file or directory.

The implementation shall support all of the variables listed in the table "Configurable Pathname Variables" and may support others. The variables in the table come from <limits.h> or <unistd.h> and the symbolic constants, defined in <unistd.h>, that are the corresponding values used for *name*.

Configurable Pathname Variables

Variable	*name* Value	Notes
{LINK_MAX}	{_PC_LINK_MAX}	1
{MAX_CANON}	{_PC_MAX_CANON}	2
{MAX_INPUT}	{_PC_MAX_INPUT}	2
{NAME_MAX}	{_PC_NAME_MAX}	3, 4
{PATH_MAX}	{_PC_PATH_MAX}	4, 5
{PIPE_BUF}	{_PC_PIPE_BUF}	6
{_POSIX_CHOWN_RESTRICTED}	{_PC_CHOWN_RESTRICTED}	7
{_POSIX_NO_TRUNC}	{_PC_NO_TRUNC}	3, 4
{_POSIX_VDISABLE}	{_PC_VDISABLE}	2

The following notes apply to the entries in the table:

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* refers to a FIFO, or *filedes* refers to a pipe or FIFO, the value returned applies to the referenced object 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* refer to a directory, the value returned applies to any files defined in this standard, other than directories, that exist or can be created within the directory.

RETURN VALUE

If *name* is an invalid value, the `pathconf()` and `fpathconf()` functions shall return −1.

If the variable corresponding to *name* has no limit for the path or file descriptor, the `pathconf()` and `fpathconf()` functions shall return −1 without changing `errno`.

If the implementation needs to use *path* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *path*, or if the process did not have the appropriate privileges to query the file specified by *path*, or *path* does not exist, the `pathconf()` function shall return −1.

If the implementation needs to use *fildes* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *fildes*, or if *fildes* is an invalid file descriptor, the `fpathconf()` function shall return −1.

Otherwise, the `pathconf()` and `fpathconf()` functions return the current variable value for the file or directory without changing `errno`. The value returned shall not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's `<limits.h>` or `<unistd.h>`.

DIAGNOSTICS

If any of the following conditions occur, the `pathconf()` and `fpathconf()` functions shall return −1 and set `errno` to the corresponding value:

EINVAL	The value of *name* is invalid.

For each of the following conditions, if the condition is detected, the `pathconf()` function shall return −1 and set `errno` to the corresponding value:

EACCES	Search permission is denied for a component of the path prefix.
EINVAL	The implementation does not support an association of the variable name with the specified file.
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 or the *path* argument points to an empty string.
ENOTDIR	A component of the path prefix is not a directory.

For each of the following conditions, if the condition is detected, the `fpathconf()` function shall return −1 and set `errno` to the corresponding value:

EBADF	The *fildes* argument is not a valid file descriptor.
EINVAL	The implementation does not support an association of the variable name with the specified file.

COPYRIGHTS

Portions of this text are reprinted from IEEE Std 1003.1-1988, *Portable Operating*

System Interface for Computer Environment, copyright © 1988 by the Institute of Electrical and Electronics Engineers, Inc., with the permission of the IEEE Standards Department. To purchase IEEE Standards, call 800/678-IEEE.

In the event of a discrepancy between the electronic and the original printed version, the original version takes precedence.

STANDARDS

In addition to the configurable pathname variables listed above, the following variables are defined in `<sys/m88kbcs.h>`:

_PC_BLKSIZE	Get optimum block size (in bytes) for I/O operations on the file, or 0 if such information is not available.

SEE ALSO

sysconf(2).

NAME

`pause` – suspend process until a signal is caught

SYNOPSIS

`int pause(void)`

DESCRIPTION

`Pause` suspends the calling process until it is presented with a signal. The signal must be one that is not currently set to be ignored by the calling process.

Neither the presentation of signals that are ignored, nor the presentation of signals that cause the termination of the calling process, nor the existence of pended signals cause pause to return.

When the signal is caught by the calling process and control is returned from the signal handler, pause returns.

ACCESS CONTROL

None.

RETURN VALUE

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to the following error code:

EINTR A signal interrupted the pause operation.

SEE ALSO

`alarm`(2), `kill`(2), `signal`(2), `wait`(2).

STANDARDS

When using `m88kbcs` as the Software Development Environment target, the `pause` function will be emulated using BCS system calls. Since this is an emulation requiring several BCS system calls, a slight performance degradation may be noticed in comparison to using `pause` in `/lib/libc.a`.

NAME

pipe – create an interprocess channel

SYNOPSIS

```
int  pipe (fildes)
int  fildes[2];
```

where:

fildes Address of an array of two file descriptors

DESCRIPTION

pipe creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. The files associated with *fildes*[0] and *fildes*[1] are streams and are both opened for reading and writing. The O_NDELAY and O_NONBLOCK flags are cleared.

A read from *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out (FIFO) basis and a read from *fildes*[1] accesses the data written to *fildes*[0] also on a FIFO basis.

The FD_CLOEXEC flag will be clear on both file descriptors.

If the process exceeds its limit for open files, the call will fail.

Pipes exist in the channel store, but not in the file name store or the flat file store. Pipes have no file attributes except time-last-accessed, time-last-changed, time-last-modified, and size.

ACCESS CONTROL

None.

RETURN VALUE

0 The pipe was successfully created.

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EFAULT The *fildes*[] buffer is an invalid area of the process's address space.

EMFILE Pipe will fail if NOFILE–1 or more file descriptors are currently open.

ENFILE The system file table is full.

SEE ALSO

sh(1), **fork**(2), **read**(2), **readv**(2), **write**(2), **writev**(2).

NAME

plock – lock data, text, or both into memory

SYNOPSIS

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

int plock(int command);
```

where:

command The specific operation to be performed

DESCRIPTION

The behavior of this call is implementation dependent. Its only effects are on performance, both of the process and of the system.

The `plock()` function allows the calling process to lock its text segment, its data segment, or both into primary memory. Locking a segment via `plock()` has no real effect in this implementation. True memory locking support is available via the `memcntl`(2) page locking operations.

The semantics of `plock` depend upon the value of *command* as follows:

`TXTLOCK` Lock text segment into memory. An error is returned and no change is made if the text segment is already locked.

`DATLOCK` Lock data segment into memory. An error is returned and no change is made if the data segment is already locked.

`PROCLOCK` Lock text and data segments into memory. An error is returned and no change is made if either the text or data segments are already locked.

`UNLOCK` Remove locks. This single operation unlocks all currently locked segments – text, data, or both. An error is returned and no change is made if neither text nor data is locked.

Note that a `TXTLOCK` and a `DATLOCK` operation, in either order, are equivalent to a `PROCLOCK` operation.

Locks are not inherited across a `fork`(2) or `vfork`(2).

ACCESS CONTROL

The effective user id of the calling process must be superuser.

RETURN VALUE

Upon successful completion, `plock()` returns a value of `0`. Otherwise, it returns the value `-1`, and sets `errno` to indicate an error.

DIAGNOSTICS

Under the following conditions, `plock()` fails and sets `errno` to:

`EPERM` if the effective user id of the calling process is not superuser.

`EINVAL` if *command* is not a valid command.

`EINVAL` if `TXTLOCK` is specified and a text lock already exists on the calling process.

`EINVAL` if `DATLOCK` is specified and a data lock already exists on the calling process.

`EINVAL` if `PROCLOCK` is specified and a text lock or a data lock already exists on the calling process.

`EINVAL` if `UNLOCK` is specified and neither a text nor a data lock exists on the calling process.

SEE ALSO

`memcntl`(2).

NAME

`poll` – input/output multiplexing

SYNOPSIS

```
#include <poll.h>
#include <stropts.h>

int  poll (poll_descriptor_array, array_size, timeout)
struct pollfd  poll_descriptor_array[];
unsigned long  array_size;
int  timeout;
```

where:

poll_descriptor_array	An array of `pollfd` structures describing the files and conditions to be checked. On output, the conditions that are actually true are filled in.
array_size	The number of entries in the array
timeout	A value specifying the timeout interval

DESCRIPTION

`poll` provides users with a mechanism for multiplexing input/output over a set of file descriptors that reference open files. `poll` identifies those files on which a user can read or write data, or on which certain events have occurred.

poll_descriptor_array specifies the file descriptors to be examined and the events of interest for each file descriptor. It is a pointer to an array with one element for each open file descriptor of interest. The array's elements are `pollfd` structures, which contain the following members:

fd	A file descriptor to an open file.
events	A flag word describing the conditions for which the stream is being checked.
revents	Ignored on input. On output, this flag word reports the conditions that have been true at some time since the start of the system call.

`fd` specifies an open file descriptor and `events` and `revents` are bitmasks constructed by an OR of any combination of the following event flags:

`POLLIN`	Data other than high priority data may be read without blocking. For STREAMS, this flag is set even if the message is of zero length.
`POLLRDNORM`	Normal data (priority band = 0) may be read without blocking. For STREAMS, this flag is set even if the message is of zero length.
`POLLRDBAND`	Data from a non-zero priority band may be read without blocking. For STREAMS, this flag is set even if the message is of zero length.
`POLLPRI`	High priority data may be received without blocking. For STREAMS, this flag is set even if the message is of zero length.
`POLLOUT`	Normal data may be written without blocking.
`POLLWRNORM`	The same as `POLLOUT`.

POLLWRBAND	Priority data (priority band > 0) may be written. This event only examines bands that have been written to at least once.
POLLMSG	An M_SIG or M_PCSIG message containing the SIGPOLL signal has reached the front of the stream head read queue.
POLLERR	An error has occured on the device or stream. This flag is only valid in the revents bitmask; it is not used in the events field.
POLLHUP	A hangup has occurred on the stream. This event and POLLOUT are mutually exclusive; a stream can never be writable if a hangup has occurred. However, this event and POLLIN, POLLRDNORM, POLLRDBAND, or POLLPRI are not mutually exclusive. This flag is only valid in the revents bitmask; it is not used in the events field.
POLLNVAL	The specified fd value does not belong to an open file. This flag is only valid in the revents field; it is not used in the events field.

For each element of the array pointed to by *poll_descriptor_array*, poll examines the given file descriptor for the event(s) specified in events. The number of file descriptors to be examined is specified by *array_size*.

If the value fd is less than zero, events is ignored and revents is set to 0 in that entry on return from poll .

The results of the poll query are stored in the revents field in the pollfd structure. Bits are set in the revents bitmask to indicate which of the requested events are true. If none are true, none of the specified bits are set in revents when the poll call returns. The event flags POLLHUP, POLLERR, and POLLNVAL are always set in revents if the conditions they indicate are true; this occurs even though these flags were not present in events. Note that the remaining conditions are not guaranteed to be true when the system call returns. All of those conditions that have been true since the start of the call are reported.

If none of the defined events have occurred on any selected file descriptor, poll waits at least *timeout* milliseconds for an event to occur on any of the selected file descriptors. If the value *timeout* is 0, poll returns immediately. If the value of *timeout* is INFTIM (or −1), poll blocks until a requested event occurs or until the call is interrupted. poll is not affected by the O_NDELAY and O_NONBLOCK flags. Poll does not wait for the full timout interval to elapse if one of the reportable conditions becomes true.

ACCESS CONTROL

None.

RETURN VALUE

0	Poll timed out and none of the reportable conditions are true on the streams of interest.
1 ... *array_size*	The number of streams for which one or more conditions are reported.
−1	The poll failed. errno indicates the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EAGAIN Memory was not available to do the `poll`.

EFAULT The poll descriptor array did not lie entirely within the caller's readable and writable address space.

EINTR A signal was caught during the `poll` call.

EINVAL *array_size* is less than zero or greater than the configured number of file descriptors.

SEE ALSO

`getmsg`(2), `putmsg`(2), `select`(2).

NAME

`profil` – set up execution time profiling for a process

SYNOPSIS

```
#include <unistd.h>

void  profil (buff, bufsiz, offset, scale)
short  * buff;
int    bufsiz;
void   (* offset)();
int    scale;
```

where:

buff — A pointer to the profiling buffer, an array of bytes in the user's address space

bufsiz — The number of bytes in the profiling buffer

offset — The offset by which the profiling program counter (PC) is adjusted before being multiplied by *scale*

scale — A value by which the PC is multiplied before indexing into the buffer array

DESCRIPTION

After the `profil` call, the user's program counter (PC) is examined at each clock tick. The value of *offset* is subtracted from the PC, and the result is multiplied by *scale*. If the resulting number corresponds to an entry in *buff*, that entry is incremented. An entry is defined as a series of bytes with length equal to `sizeof(short)`.

Scale is interpreted as an unsigned, fixed-point number with the binary point 16 bits from the right. For a machine whose instructions are 32 bits in size, such as the MC88000, 0x8000 gives a 1–1 mapping of instructions to entries in *buff*; 0x4000 maps each pair of instructions together, etc.

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when you call exec(2) but remains on in both the child and parent after a call to `fork`(2). Profiling will be turned off if an update in *buff* would cause a memory fault.

RETURN VALUE

None.

DIAGNOSTICS

None.

SEE ALSO

exec(2).

NAME

ptrace – process trace

SYNOPSIS

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

int  ptrace (request, pid, address, data)
int    request;
pid_t  pid;
int    address;
int    data;
```

where:

request	Process trace command
pid	Process being traced (used only if *request* is 1-8)
address	Optional address argument (used only if *request* is 1-7)
data	Optional data argument (used only if *request* is 4-7)

DESCRIPTION

Ptrace lets a process (debugger process) control the execution of another process (target process). Its primary use is to implement breakpoint debugging; see sdb(1) and dbx(1). The target process behaves normally until it encounters a signal (see sys/signal.h for the list) or until it exits; it then stops for tracing and its debugger process is notified via wait. (A signal that is blocked does not cause the process to stop for tracing until it is unblocked.) When the target process is stopped, its debugger process can examine and modify its core image using ptrace. Also, the debugger process can cause the target process either to terminate or continue, with the possibility of ignoring the signal that caused it to stop. If the debugger process terminates while tracing a target, the target will be sent a SIGKILL signal. While a process is being traced via ptrace(2), job control stop signals are ignored.

The normal sequence of events required to trace a child process is as follows:

1. The child process is created by the fork operation.
2. The child process performs a ptrace operation with *request* set to 0.
3. The child's address space is changed by the exec operation. This causes the child to be stopped before executing the first instruction of the new image as if the signal SIGTRAP had occurred.
4. The parent process waits for the child to stop using a wait operation.
5. The parent may now cause the child to continue execution using ptrace with *request* set to 7.

The normal sequence of events required to trace a non-child process is as follows:

1. The controlling process is created by the fork operation.
2. The controlling process performs a ptrace operation with *request* set to 128. If the target process is stopped due to a job control signal (e.g., SIGSTOP) at the time *request* 128 is issued, the ptrace call will complete normally but tracing does not actually occur until the target process leaves the stopped state (due to a signal that continues or terminates it).
3. The controlling process waits for the target process to stop using a wait operation.

4. The controlling process may now cause the target process to continue execution using `ptrace` with *request* set to 7.

The *request* argument determines the precise action to be taken by `ptrace` and is one of the following:

0 The child process must issue this request if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by its signal handler. The *pid*, *address*, and *data* arguments are ignored, and a return value is not defined for this request. (Unexpected results may ensue if the parent does not expect to trace the child. The parent may not cause the child to continue after a signal, and the child will be terminated if the parent terminates.)

The other requests can be used only by the controlling process. For each, *pid* is the process id of the target, and *address* is a user address. The offset is a word address. The target must have stopped for tracing before these requests are made otherwise, the error condition ESRCH is asserted.

1 or 2 With these requests, the word at location *address* in the address space of the target is returned to the controlling process. The *data* argument is ignored. These two requests will fail if *address* is not a valid word pointer, in which case the error condition EIO is asserted and a −1 is returned.

3 With this request, information about the target process stored in the kernel address space is made available to the controlling process. This information is referenced by *addr* which is interpreted as a word offset into a synthetic `ptrace_user` structure (see `sys/user.h`). The *data* argument is ignored. The request will fail if *addr* is not a relative word offset within the `ptrace_user` structure or if *addr* is not a valid word * offset, in which case the error condition EIO is asserted and a −1 is returned.

4 or 5 With these requests, the 32-bit value given by the *data* argument is written into the address space of the target at location *address*. Upon successful completion, the value is returned to the controller. These two requests will fail if *address* is a location in a pure procedure space and another process is executing in that space, or if *address* is not a valid word pointer. Upon failure the error condition EIO is asserted and a −1 is returned. Upon successful completion, the value written into the address space will be returned.

6 With this request, information about the target process stored in the kernel may be changed. This is similar to request 3 above, but only a few entries in the `ptrace_user` structure may be changed (see `sys/user.h`). *Data* gives the value that is to be written and *address* is the word offset of the entry.

7 This request causes the target to resume execution. If the *data* argument is a valid signal number, the target resumes execution as if it had incurred that signal, and any other pending signals are cancelled. The *address* argument must be equal to 1 for this request. Upon successful completion, the value of *data* is returned to the controlling process. This request will fail if *data* is not 0 or a valid signal number, in which case the error condition is asserted and a −1 is returned.

8 This request causes the target to terminate with the same consequences as exit, except that the target does not stop for tracing again as part of exiting.

9 Single step through the instructions in the target process.

128 The controlling process initiates a debugging session with an existing process whose process id is *pid*.

129 The controlling process terminates a debugging session with a given process. If *addr* is not 1, it becomes the new program counter of the (ex)target process.

130 Any forked children of the target process will inherit their parent's debugger and trace state.

To forestall possible fraud, `ptrace` inhibits the set-user-id facility on subsequent `exec` calls. If a traced process calls `exec`, it will stop before executing the first instruction of the new image showing signal SIGTRAP.

ACCESS CONTROL

None.

RETURN VALUE

For *request* values of 0, 6, 8, 128, 129, or 130, the following values are returned:

0 The particular request was successful.

−1 An error occurred. `errno` is set to indicate the error.

For *request* values of 1, 2, or 3, the following values are returned:

value The 32-bit value read from the given target address. This value may be −1.

−1 An error occurred. `errno` is set to indicate the error.

For *request* values of 4, 5, 7, or 9, the following values are returned:

data The value of *data* is returned.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EIO *Request* is an illegal number.

ESRCH *pid* identifies a child that does not exist or has not executed a `ptrace` with *request* 0.

SEE ALSO

`exec(2)`, `signal(2)`, `wait(2)`.

NAME

putmsg, putpmsg – pass a message down a stream

SYNOPSIS

```
#include <stropts.h>

int  putmsg(filedes, control_info_ptr, data_info_ptr, flags)
int  filedes;
struct strbuf * control_info_ptr;
struct strbuf * data_info_ptr;
int  flags;

int  putpmsg(filedes, control_info_ptr, data_info_ptr, band, flags)
int  filedes;
struct strbuf * control_info_ptr;
struct strbuf * data_info_ptr;
int  band;
int  flags;
```

where:

filedes	A valid, active descriptor referring to an open streams file
control_info_ptr	A pointer to a structure describing the control buffer or NULL, if there is no control buffer
data_info_ptr	A pointer to a structure describing the data buffer or NULL, if there is no data buffer
band	The priority band the message is to be sent in.
flags	Indicates the type of message to be sent.

DESCRIPTION

putmsg creates a message from user-specified buffer(s) and sends the message to a STREAMS file. The message may contain either a data part, a control part, or both. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The semantics of each part is defined by the STREAMS module that receives the message.

The function putpmsg does the same thing as putmsg, but provides the user the ability to send messages in different priority bands. Except where noted, all information pertaining to putmsg also pertains to putpmsg.

fd specifies a file descriptor referencing an open stream. *ctlptr* and *dataptr* each point to a strbuf structure, which contains the following members:

buf Pointer to the first byte of the control or data information.

len The number of bytes of information in the buffer.

maxlen Ignored [see getmsg(2)].

To send the data part of a message, *data_info_ptr* must not be NULL and the len field of *data_info_ptr* must have a value of 0 or greater. To send the control part of a message, the corresponding values must be set for *control_info_ptr*. No data (control) part is sent if either *data_info_ptr (control_info_ptr)* is NULL or the len field of *data_info_ptr (control_info_ptr)* is set to –1.

For putmsg(), if a control part is specified, and *flags* is set to RS_HIPRI, a high priority message is sent. If no control part is specified, and *flags* is set to RS_HIPRI, putmsg fails and sets errno to EINVAL. If *flags* is set to 0, a normal (non-priority) message is sent. If no control part and no data part are specified, and *flags* is set to 0, no message is sent, and 0 is returned.

The stream head guarantees that the control part of a message generated by `putmsg` is at least 64 bytes in length.

For `putpmsg`, the flags are different. *flags* is a bitmask with the following mutually-exclusive flags defined: `MSG_HIPRI` and `MSG_BAND`. If *flags* is set to 0, `putpmsg` fails and sets `errno` to `EINVAL`. If a control part is specified and *flags* is set to `MSG_HIPRI` and *band* is set to 0, a high-priority message is sent. If *flags* is set to `MSG_HIPRI` and either no control part is specified or *band* is set to a non-zero value, `putpmsg()` fails and sets `errno` to `EINVAL`. If *flags* is set to `MSG_BAND`, then a message is sent in the priority band specified by *band*. If a control part and data part are not specified and *flags* is set to `MSG_BAND`, no message is sent and 0 is returned.

Normally, `putmsg()` will block if the stream write queue is full due to internal flow control conditions. For high-priority messages, `putmsg()` does not block on this condition. For other messages, `putmsg()` does not block when the write queue is full and `O_NDELAY` or `O_NONBLOCK` is set. Instead, it fails and sets `errno` to `EAGAIN`.

`putmsg` or `putpmsg` also block, unless prevented by lack of internal resources, waiting for the availability of message blocks in the stream, regardless of priority or whether `O_NDELAY` or `O_NONBLOCK` has been specified. No partial message is sent.

ACCESS CONTROL

Fildes must be open for writing.

RETURN VALUE

0 The message was successfully sent.

−1 The message was not sent. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EAGAIN The O_NDELAY or O_NONBLOCK flag was set, a non-priority message was specified, and the stream write queue is full due to internal flow control conditions; or streams buffers could not be allocated for the message.

EBADF *Fildes* is not a valid, active descriptor open for writing.

EFAULT The arguments pointed to by *control_info_ptr*, or *data_info_ptr* do not lie entirely within the caller's readable address space.

EINTR A signal was caught during the `putmsg` call.

EINVAL An illegal value was specified by *flags* or *flags* was RS_HIPRI and there was no control part of the message; or the stream referred to by *fildes* is linked under a multiplexor.

ENXIO A hangup condition was generated downstream for the specified stream.

ERANGE The size of the data part of the message does not fall within the range specified by the minimum and maximum packet sizes of the write side of the topmost module on the stream; or the control or data part of the message exceeded the configured maximum for that part of a message.

ENOSR If a stream is not associated with *filedes*.

SEE ALSO

getmsg(2), poll(2).

NOTE

The user should avoid using O_NDELAY and instead should use O_NONBLOCK.

NAME

`read` – read from an object

SYNOPSIS

```
int       read (fildes, buffer, nbyte)
int       fildes;
char      buffer[];
unsigned  nbyte;
```

where:

fildes An active, valid file descriptor.

buffer User data buffer.

nbyte Size (in bytes) of the user data buffer.

DESCRIPTION

`Read` transfers *nbyte* bytes of data from the object associated with *fildes* into the buffer pointed to by *buffer*.

If *fildes* refers to an object pointer having a current position attribute, the read starts at a position in the object given by that attribute. If the current position refers to a part of a file that has never been written (i.e., a part of a file that was created by seeking past the end of the file) then the value of the data is all zeros.

If the object pointer has no position attribute, then the starting read position depends on the type of object being read.

The behavior of the `read` call is affected by the object attribute flag O_NDELAY (see `open`(2)) associated with *fildes*.

If the O_NDELAY flag is set and *fildes* refers to a file that has mandatory record locking enabled and is currently write locked, the call returns −1 and `errno` is set to EAGAIN. If O_NDELAY is clear, the call blocks until the appropriate lock is removed or the call is interrupted by a signal.

When attempting to read from an empty pipe (or fifo) the following will occur: If no process has the pipe open for writing, 0 is returned to indicate end-of-file. If some process has the pipe open for writing, and O_NDELAY is set, 0 is returned. If some process has the pipe open for writing, and O_NONBLOCK is set, −1 is returned and `errno` is set to EAGAIN. If some process has the pipe open for writing, and O_NDELAY is clear, the call will block until some data is written or the pipe is closed by all processes that had opened the pipe for writing.

When attempting to read a file associated with a character special file that has no data currently available the following will occur: If O_NDELAY is set, −1 is returned and `errno` is set to EAGAIN. If O_NDELAY is clear, the call will block until some data becomes available.

A `read` from a STREAMS [see `intro`(2)] file can operate in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the `I_SRDOPT` `ioctl`(2) request [see `streamio`(7)], and can be tested with the `I_GRDOPT` `ioctl`(2) request. In byte-stream mode, `read` usually retrieve data from the stream until they have retrieved *nbyte* bytes, or until there is no more data to be retrieved. Byte-stream mode usually ignores message boundaries.

In STREAMS message-nondiscard mode, `read` retrieves data until they have read *nbyte* bytes, or until they reach a message boundary. If `read` does not retrieve all the data in a message, the remaining data is replaced on the stream and can be retrieved by the next `read` call. Message-discard mode also retrieves data until it has retrieved *nbyte* bytes, or it reaches a message boundary. However, unread data remaining in a message after the `read` returns is discarded, and is not available for a subsequent `read` or `getmsg` [see `getmsg`(2)].

When reading from a STREAMS file, handling of zero-byte messages is determined by the current read mode setting. In byte-stream mode, `read` accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. `read` then returns the number of bytes read, and places the zero-byte message back on the stream to be retrieved by the next `read` or `getmsg` [see `getmsg`(2)]. In the two other modes, a zero-byte message returns a value of 0 and the message is removed from the stream. When a zero-byte message is read as the first message on a stream, a value of 0 is returned regardless of the `read` mode.

A `read` from a STREAMS file returns the data in the message at the front of the stream head read queue, regardless of the priority band of the message.

Normally, a `read` from a STREAMS file can only process messages with data and without control information. The `read` fails if a message containing control information is encountered at the stream head. This default action can be changed by placing the stream in either control-data mode or control-discard mode with the `I_SRDOPT` `ioctl`(2). In control-data mode, control messages are converted to data messages by `read`. In control-discard mode, control messages are discarded by `read`, but any data associated with the control messages is returned to the user.

When `read` completes, the position attribute, if it exists, is incremented by the number of bytes actually read. The access time for the file is updated to reflect the time the read occurred, unless the file resides on a read-only file system.

If an error occurs, the contents of *buffer* and any changes to the object associated with *fildes* are defined by the object's type. The default situation is that *buffer* and the object associated with *fildes* are unchanged. This may not be the case for some errors on some types of objects.

ACCESS CONTROL

Fildes must be open for reading.

RETURN VALUE

0..*nbyte*	Completed successfully. The number of bytes actually read is returned. The value 0 indicates the 'end-of-file' condition.
−1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	*Fildes* is not a valid file descriptor open for reading.
EAGAIN	O_NDELAY is set on the I/O channel and there is a mandatory lock on the file owned by a different process.
EAGAIN	A read was attempted on an empty pipe that another process has open for writing.
EAGAIN	A read was attempted on an I/O channel that had O_NDELAY set, but there was no data ready to be read at the time of the call.
EFAULT	*Buffer* points outside the allocated address space.
EINTR	A signal was caught during the system call.

EDEADLK *fildes* refers to a file that has mandatory record locking enabled and the read would produce a deadlock condition. See `lockf`(2) for a discussion of deadlock conditions.

SEE ALSO

creat(2), **dup**(2), **dup2**(2), **fcntl**(2), **ioctl**(2), **open**(2), **pipe**(2), **readv**(2), **select**(2), **socket**(2), **socketpair**(2), **termio**(7).

NAME

`readlink` – read the contents of a symbolic link

SYNOPSIS

```
#include <unistd.h>

int  readlink (path, buffer, nbyte)
char * path;
char * buffer;
int  nbyte;
```

where:

path	Address of a pathname naming a symbolic link
buffer	User data buffer
nbyte	Size (in bytes) of the user data buffer

DESCRIPTION

`Readlink` reads at most the first *nbytes* of the symbolic link file into the buffer pointed to by *buffer*. The last component of *path* is a symbolic link file, and the pathname resolution does not follow the symbolic link.

A terminating null character is not added to the end of the link contents (or to the end of the buffer, should the buffer size be less than the size of the symbolic link file). Hence, readlink's return value, the number of characters placed in the buffer, is the only clue the process has to how much of *buffer* contains valid data.

If `readlink` fails, the contents of the buffer are undefined.

ACCESS CONTROL

The calling process must have permission to resolve *path*.

RETURN VALUE

nbyte	Completed successfully. The number of characters placed in the buffer is returned. No determination can be made as to whether the entire contents of the symbolic link file have been read.
0..*nbyte−1*	Completed successfully. The number of characters placed in the buffer is returned.
−1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EFAULT	*Buffer* extends outside the process's allocated address space.
ENOENT	The named file does not exist.
EINVAL	The named file is not a symbolic link.
EPERM	Permission to read the symbolic link is denied to the calling process.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.

ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.

SEE ALSO

`lstat`(2), `stat`(2), `symlink`(2).

NAME

readv – read from file

SYNOPSIS

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

int     readv (fildes, iov, iovcnt)
int    fildes;
struct iovec iov[];
int    iovcnt;
```

where:

fildes An active, valid file descriptor

iov An array of extents

iovcnt The number of extents given

DESCRIPTION

Readv transfers data from the object associated with *fildes* into the *iovcnt* buffers specified by the members of *iov*[]: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*–1]. Each *iov*[] member specifies the base address and length of an area in memory where data should be placed. Readv fills an area completely before proceeding to the next.

The *iov* structure is defined as:

```
struct iovec {
caddr_t  iov_base;
int      iov_len;
};
```

Iovcnt must be a positive number less than or equal to a system-imposed limit guaranteed to be at least MAXIOVCNT. The length of each extent (iov_len) in *iov*[] must be non-negative and the sum of these lengths must not overflow a 'long'.

Except for the disposition of the data, readv is equivalent to read.

ACCESS CONTROL

Fildes must be open for reading.

RETURN VALUE

0..nbyte Completed successfully. The number of bytes actually read is returned. The value 0 indicates the 'end-of-file' condition. Here, nbyte is the sum of the lengths of the *iovcnt* extents given in *iov*[].

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

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

EAGAIN O_NDELAY is set on the I/O channel and there is a mandatory lock on the file owned by a different process.

EAGAIN A read was attempted on an empty pipe that another process has open for writing.

EAGAIN A read was attempted on an I/O channel that had O_NDELAY set, but there was no data ready to be read at the time of the call.

EFAULT *Iov* points outside the allocated address space.

EFAULT	One or more of the *iov* [] members point outside the allocated address space.
EINTR	A signal was caught during the system call.
EINVAL	*Iovcnt* was invalid.
EINVAL	One or more of the iov_len values in *iov* [] was negative.
EINVAL	The sum of the iov_len values in *iov* [] overflowed a 'long'.

SEE ALSO

`creat`(2), `dup`(2), `dup2`(2), `fcntl`(2), `ioctl`(2), `open`(2), `pipe`(2), `read`(2), `select`(2), `socket`(2), `socketpair`(2), `termio`(7).

NAME

`reboot` – reboot halts and optionally reboots the system processor(s)

SYNOPSIS

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

int  reboot (howto)
int  howto;
```

where:

howto A mask of options specifying the type of shutdown to perform

DESCRIPTION

The `reboot` system call halts the system processor(s). The *howto* mask specifies the type of shutdown to perform. The possible values of *howto* are:

RB_HALT — The processor(s) is (are) simply halted. Use with caution.

RB_SHUTDOWN — The system is shut down and the processor(s) is (are) halted. All user processes are killed, and the buffer cache is flushed.

RB_AUTOBOOT — The system is shut down and the processor(s) is (are) halted. All user processes are killed, and the buffer cache is flushed. The system is then rebooted using the current boot path (the default is the boot path used when the system was last booted). Use the `dg_sysctl`(2) system call to alter the current boot path.

ACCESS CONTROL

Only the super-user may halt the system processor(s).

RETURN VALUE

If successful, this call never returns. Otherwise, a –1 is returned, and `errno` is set to return the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EPERM — The caller is not super-user.

EINVAL — The option specified in *howto* was not RB_HALT or RB_SHUTDOWN.

SEE ALSO

`dg_sysctl`(1M), `halt`(1M), `reboot`(1M), `dg_sysctl`(2).

NAME

`recv` – receive a message from a socket

SYNOPSIS

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

int  recv (s, buf, len, user_flags)
int  s;
char * buf;
int  len;
int  user_flags;
```

where:

s	File descriptor of socket to receive data from
buf	Buffer for data
len	Length of buffer for data (bytes)
user_flags	Flags for transfer

DESCRIPTION

This call can be used only with connected sockets.

If the message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from; datagram sockets truncate messages, stream sockets don't preserve packet boundaries so only the amount of data requested is received with no loss of data.

The *user_flags* argument is constructed by or-ing zero or more literals beginning with "MSG_". See `<sys/socket.h>` for a description of what flags exist and what they do.

If no messages are available at the socket, the call waits for a message to arrive, unless the socket is nonblocking [see `ioctl`(2)]. In that case, an error EAGAIN will be returned.

The `select` call may be used to determine when more data arrives.

ACCESS CONTROL

None.

RETURN VALUE

`Recv` returns the number of bytes received.

0..*len*	Number of bytes transferred.
–1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	The argument *s* is not an active valid descriptor.
ENOTSOCK	The argument *s* is not a socket.
EAGAIN	The socket is marked non-blocking and the receive operation would block.
EINTR	The receive was interrupted by delivery of a signal before any data was available for the receive.
EFAULT	The data was specified to be received into a non-existent or protected part of the process address space.

EINVAL	Invalid argument.
ENOTCONN	The socket is not connected. Use `recvfrom`(2).
EOPNOTSUPP	The *flags* argument included the MSG_OOB flag applied to a UDP socket.

SEE ALSO

`ioctl`(2), `read`(2), `recvfrom`(2), `select`(2), `send`(2), `socket`(2).

NAME

`recvfrom` – receive a message from a socket

SYNOPSIS

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

int  recvfrom (s, buf, len, user_flags, from, fromlen)
int  s;
char * buf;
int  len;
int  user_flags;
struct sockaddr *from;
int * fromlen;
```

where:

s	File descriptor of socket to receive a message from
buf	Buffer for message
len	Length of buffer
user_flags	Flags for transfer
from	Structure to hold sender's name
fromlen	On input contains the number of bytes available for the sender's name; updated to indicate the number of bytes returned

DESCRIPTION

The `recvfrom` call is identical to the `recv` call with the addition of returning the name of the socket from which the message was sent. When using `recvfrom` on connected sockets, the *from* and *fromlen* arguments will be undefined. When reading from datagram sockets, messages that are longer than the buffer are truncated. See `recv`(2) for additional information about the socket receive mechanism.

ACCESS CONTROL

None.

RETURN VALUE

This call returns the number of bytes received.

1..*len*	Number of bytes transferred.
–1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	The argument *s* is not an active valid descriptor.
ENOTSOCK	The argument *s* is not a socket.
EAGAIN	The socket is marked non-blocking and the receive operation would block.
EINTR	The receive was interrupted by delivery of a signal before any data was available for the receive.
EFAULT	The data was specified to be received into a non-existent or protected part of the process address space.
EINVAL	Bad argument.

SEE ALSO

`read`(2), `recv`(2), `send`(2), `socket`(2).

NAME

recvmsg – receive a message from a socket

SYNOPSIS

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

int     recvmsg (s, msg, user_flags)
int     s;
struct msghdr * msg;
int     user_flags;
```

where:

s	File descriptor of socket to receive from
msg	Pointer to receive msg packet
user_flags	Flags for transfer

DESCRIPTION

The recvmsg call is identical to recv or recvfrom depending on whether or not the *msg_namelen* fields are greater than zero. If the *msg_namelen* field of the msghdr structure is non-zero, this call is identical to recvfrom, otherwise it is identical to recv.

The added value of this call is that it allows an IOV to be supplied in the msg packet for use of non-contiguous buffers (see readv for more information about IOV structures).

ACCESS CONTROL

None.

RETURN VALUE

These calls return the number of bytes received.

0..*len*	Number of bytes transferred.
–1	An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EBADF	The argument *s* is not an active valid descriptor.
ENOTSOCK	The argument *s* is not a socket.
EAGAIN	The socket is marked non-blocking and the receive operation would block.
EINTR	The receive was interrupted by delivery of a signal before any data was available for the receive.
EFAULT	The data was specified to be received into a non-existent or protected part of the process address space.
EMSGSIZE	Too many entries in the iovec array.

SEE ALSO

read(2), readv(2), recvfrom(2), send(2), socket(2).

NAME

`rename` – change the name of a file

SYNOPSIS

```
int  rename (old_path, new_path)
char * old_path;
char * new_path;
```

where:

old_path Address of the pathname of the file being renamed

new_path Address of file's new pathname

DESCRIPTION

Old_path points to a pathname naming an existing file that will be called the source file. *New_path* points to a pathname naming a target file that may or may not exist. If the target exists, it must be the same type as the source file. In either case, the source and target must reside on the same file system device. '.' and '..' cannot be renamed. Terminal symbolic links for either pathname are not followed.

If both files are directories, *old_path* must not be an ancestor of *new_path*. This prevents the rename operation from orphaning everything in the file hierarchy below *old_path*. If *new_path* is an existing directory, it must contain no entries but '.' and '..', and the only links to it should be its '.' entry and its entry in its parent.

The link between the pathname *old_path* and the source file is deleted, though there may be other links to the source file. If the target file exists, the link between *new_path* and the target is also deleted. Lastly, a link between the pathname *new_path* and the source file is created. This sequence of events is described in more detail below:

If *new_path* does not exist in the filename store, a link for it is created in the directory indicated by the path prefix of *new_path*. The link is made to refer to the same entity in the filesystem that *old_path* refers to.

If *new_path* already exists in the filename store, the link in its containing directory is changed to refer to the same entity in the filesystem that *old_path* refers to. If this change deletes the last link to the file formerly referred to by *new_path*, that file is deleted.

Old_path is removed from the filename store.

The attributes of the files involved change as follows:

- Source File – The time of last attribute change (st_ctime) is set to the current time.
- Target File (if it existed and was not deleted) – The number of links (st_nlink) is decremented. The time of last attribute change (st_ctime) is set to the current time.
- Containing Directory of Source File – The time last modified (st_mtime) and time of last attribute change (st_ctime) are set to the current time. If rename is operating on directories and either the target file existed or the parent of the target file differs from the parent of the source file, the number of links (st_nlink) is decremented. The file size (st_size) is updated to reflect the deletion of the entry for *old_path* and possibly, the addition of an entry for *new_path*.
- Containing Directory of Target File (assuming it differs from the containing directory of the source file) – If rename is operating on directories and the

target file didn't exist, the number of links (st_nlink) is incremented and the time of last attribute change (st_ctime) is set to the current time. (This reflects that the '..' of the source is set to a new directory, namely, what was the parent of the target file.) The file size (st_size) is updated if the target file didn't exist, reflecting the addition of an entry for *new_path*.

If the call fails, the attributes of all files and directories are unchanged.

ACCESS CONTROL

If the source file is a directory and its parent will change, the calling process must have write access to the source in order to change its '..' entry.

The process must have write permission to the containing directories.

The process must have permission to resolve *old_path* and *new_path*.

RETURN VALUE

0	The file was successfully renamed.
−1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EPERM	The file named by *old_path* is a directory and the effective user id is not superuser.
EXDEV	The link named by *new_path* and the file named by *old_path* are on different logical devices (file systems). Note that this error code will not be returned if the implementation permits cross-device links.
EACCES	The requested link requires writing in a directory with a mode that denies write permission.
EROFS	The requested link requires writing in a directory on a read-only file system device.
EINVAL	The file named by *old_path* is an ancestor directory of the file named by *new_path*.
EINVAL	The file named by *old_path* is '.' or '..'.
EISDIR	*Old_path* is a directory and *new_path* is not.
ENOSPC	No more contiguous space for a new directory entry.
EEXIST	*New_path* points to a non-empty directory.
ENOENT	*Old_path* does not exist.
ENOENT	A non-terminal component of *old_path* or *new_path* does not exist.
ENOTDIR	A non-terminal component of *old_path* or *new_path* was not a directory or symbolic link.
ENAMETOOLONG	*Old_path* or *new_path* exceeds the length limit for pathnames.
ENAMETOOLONG	A component of *old_path* or *new_path* exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve *old_path* or *new_path* or to expand a symbolic link.

ELOOP — The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.

EPERM — *Old_path* or *new_path* contains a character not in the allowed character set.

EFAULT — *Old_path* or *new_path* does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

`mv`(1), `mvdir`(1M), `open`(2), `stat`(5).

NAME

`rmdir` – remove a directory file

SYNOPSIS

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

where:

path Address of a pathname naming an existing directory

DESCRIPTION

`Rmdir` removes the directory from the filename store. The directory named cannot be the calling process's current working directory or a directory containing a mounted file system. The directory should have no entries but '.' and '..', referring to the directory itself and its parent. There must be exactly two links to the directory – its own '.' and the entry for it in its parent. Note that precluding the removal of the working directory and non-empty directories ensures that the current root directory cannot be removed; if the root were empty, it would have to be the current working directory.

The directory is removed from the filename store by deleting the link to it in its parent.

The attributes of the parent change as follows: The number of links (st_nlink) is decremented, reflecting the fact that the '..' of the removed directory will no longer refer to the parent. The time of last attribute change (st_ctime) is set to the current time.

The attributes of the directory change as follows: Its size (st_size) and number of links (st_nlink) are set to 0. The time last modified (st_mtime) and time of last attribute change (st_ctime) are set to the current time.

Some process may have the directory open for reading at the time it is removed. Upon attempting the next read operation, that process will encounter the end-of-file condition, as the directory's size is now zero.

When the last reference to the directory is deleted (examples of references are when some process has the directory open or it is the working or root directory for a process), the directory is removed from the filesystem.

If the call fails, the directory is not removed, and the attributes of the directory and its parent are unchanged.

ACCESS CONTROL

The calling process must have write access to the parent of the directory being removed.

The process must have permission to resolve *path*.

RETURN VALUE

0 The directory was successfully deleted.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EACCES	Write permission is denied on the directory containing the link to be removed.
ENOTDIR	The file to be removed is not a directory.

EBUSY	The directory to be removed is currently in use by the system (as a mount point for a mounted file system device, or mounted on a remote system).
EINVAL	The directory to be removed is the current working directory.
EEXIST	The named directory contains files other than "." and ".." in it.
EEXIST	There are not exactly 2 links to the directory.
EROFS	The directory entry to be removed resides on a file system mounted read-only.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

`mkdir`(1), `rm`(1), `rmdir`(1), `mkdir`(2), `unlink`(2), `stat`(5).

NAME

sbrk – change data segment space allocation

SYNOPSIS

```
#include <unistd.h>

void *sbrk(int increment);
```

where:

increment The signed increment by which to change the data area size

DESCRIPTION

The `sbrk()` system call dynamically changes the amount of space allocated for the calling process's data segment; see `exec`(2). The change is made by adding *increment* to the process's current break value and allocating or deallocating the appropriate amount of space. The break value is the address of the first byte beyond the end of the data segment. The amount of allocated space increases as the break value increases. If *increment* is positive, space is allocated, and any newly allocated pages will be initialized with zero bytes; that is, if these addresses are read before they are written, the contents will be zero. If *increment* is negative, space is deallocated from the data segment. The contents of the addresses from the new break value to the prior break value become undefined.

There is a maximum possible break value for a process; this value may be obtained by calling the `ulimit`(2) function. There is also a program-dependent minimum break value for a process; this minimum is greater than or equal to the address of the first byte in the data segment, and less than or equal to the program's initial break value.

The `sbrk()` call will fail without making any change in the allocated space if an error occurs.

ACCESS CONTROL

No access check is made.

RETURN VALUE

Upon successful completion, `sbrk()` returns the previous break value. Otherwise, it returns the value `(void *) -1`, and sets `errno` to indicate an error.

DIAGNOSTICS

Under the following conditions, `sbrk()` fails and sets `errno` to:

ENOMEM	if the change would allocate more space than is allowed by a system-imposed maximum (see `ulimit`(2)).
ENOMEM	if the change would allocate more space than is allowed by the current data resource limit (see `getrlimit`(2)).
ENOMEM	if the change would make the break value greater than or equal to the start address of an attached shared memory segment (see `shmat`(2)).
EFAULT	if the change would make the break value less than the system-imposed minimum.
EAGAIN	if the change would allocate more space than the available physical memory and swap space.
EAGAIN	if the `MCL_FUTURE` memory locking option is in effect for the calling process (see `memcntl`(2)), and the system-imposed limit on space locked into physical memory would be exceeded.

SEE ALSO

`exec`(2), `getrlimit`(2), `memcntl`(2), `ulimit`(2).

NAME

`select` – wait for I/O conditions

SYNOPSIS

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

int  select (nfds, readfds, writefds, exceptfds, timeout)
int  nfds;
long * readfds;
long * writefds;
long * exceptfds;
struct timeval *timeout;
```

where:

nfds	The range of file descriptors to examine
readfds	The address of a bit mask representing file descriptors ready for reading
writefds	The address of a bit mask representing file descriptors ready for writing
exceptfds	The address of a bit mask representing file descriptors having an exception
timeout	Maximum selection duration

DESCRIPTION

`Select` examines the descriptors specified by the bit masks *readfds*, *writefds*, and *exceptfds* to see if they are ready for reading, writing, or have an exceptional condition pending, respectively.

Descriptor *f* is represented in a bit mask by the value "1<<*f*". Furthermore, only descriptors 0 through *nfds*–1 inclusive are examined.

The *timeout* parameter specifies a maximum interval to wait for a descriptor to become ready. If *timeout* is NULL, select will wait indefinitely. Otherwise, the maximum wait time is given by the value of the structure located at *timeout*.

Select returns when either the maximum wait interval has expired or at least one condition for one of the descriptors exists.

Select returns, in place, a mask of descriptors that are ready. The descriptor masks are only modified if the return value is non-negative.

ACCESS CONTROL

None.

RETURN VALUE

1..3**nfds*	Completed successfully. The sum of the number of descriptors identified in each bit mask is returned.
0	Time limit exceeded.
–1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	One of the bit masks specified an invalid descriptor.
EINTR	A signal was delivered before any of the selected-for events occurred and before the time limit expired.
EINVAL	*nfds*==0.

SEE ALSO

accept(2), connect(2), read(2), readv(2), recv(2), send(2), write(2), writev(2).

NAME

`semctl` – semaphore control operations

SYNOPSIS

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

int   semctl (semid, semnum, cmd, arg)
int   semid;
int   semnum;
int   cmd;
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
}arg;
```

where:

semid A semaphore set identifier

semnum The subject semaphore (used only if *cmd* is GETVAL, SETVAL, GETNCNT, GETZCNT or GETPID)

cmd The semaphore operation to be performed

arg An argument (used only if *cmd* is SETVAL, GETALL, SETALL, IPC_STAT, OR IPC_SET)

DESCRIPTION

`Semctl` provides semaphore control operations as specified by *cmd*. The subject semaphore set is identified by *semid*. The action taken depends on the value of *cmd* as follows:

SETVAL Set the value of semaphore number *semnum* to *arg.val*.

When this *cmd* is successfully executed, all processes having a semaphore adjustment value corresponding to semaphore number *semnum* will have those values set to zero.

If an error occurs, the semaphore set is unchanged.

GETVAL Return the value of semaphore number *semnum*.

GETPID Return the process id of the last process to perform an operation on semaphore number *semnum*.

GETNCNT Return the number of processes waiting for the value of semaphore number *semnum* to increase.

GETZCNT Return the number of processes waiting for the value of semaphore number *semnum* to become zero.

SETALL Set the value of all semaphores in the specified semaphore set to the values contained in the array pointed to by *arg.array*.

When this *cmd* is successfully executed, all processes having a semaphore adjustment value corresponding to a semaphore in the specified semaphore set will have those values set to zero.

If an error occurs, the semaphore set is unchanged.

GETALL Return the value of all semaphores in the specified semaphore set using the array pointed to by *arg.array*.

If an error occurs, the contents of *arg.array* are undefined.

IPC_STAT The current semaphore set attributes are stored in the structure pointed to by *arg.buf*.

If an error occurs, the contents of *arg.buf* are undefined.

IPC_SET The following semaphore set attributes are set to the values found in the structure pointed to by *arg.buf*: user id (sem_perm.uid), group id (sem_perm.gid), and permission rights (in sem_perm.mode).

If an error occurs, the semaphore set remains unchanged. Otherwise, the last change time (sem_ctime) is set to the current time.

IPC_RMID The semaphore set is destroyed. All resources consumed by the semaphore set are freed and the semaphore set identifier is invalidated.

If an error occurs, the semaphore set remains unchanged.

ACCESS CONTROL

Operation permission depends on the value of *cmd* as follows:

- If *cmd* is GETVAL, GETPID, GETNCNT, GETZCNT, GETALL, or IPC_STAT, the calling process is required to have read access to the semaphore set.
- If *cmd* is SETVAL or SETALL, the calling process is required to have alter access to the semaphore set.
- If *cmd* is IPC_SET or IPC_RMID, the effective user id of the calling process must be equal to the semaphore set's user id (sem_perm.uid), the semaphore set creator's user id (sem_perm.cuid), or that of the superuser.

RETURN VALUE

The value returned may be the following regardless of the value of *cmd*:

−1 An error occurred. `errno` is set to indicate the error.

If *cmd* is GETVAL, the value returned may be the following:

semaphore_value Completed successfully. The specified semaphore's value is returned.

If *cmd* is GETPID, the value returned may be the following:

process_id Completed successfully. The process id of the last process to perform an operation on the specified semaphore is returned.

If *cmd* is GETNCNT, the value returned may be the following:

process_id_count Completed successfully. The number of processes waiting for the value of the specified semaphore to increase is returned.

If *cmd* is GETZCNT, the value returned may be the following:

process_id_count Completed successfully. The number of processes waiting for the value of the specified semaphore to become zero is returned.

If *cmd* is SETVAL, GETALL, SETALL, IPC_STAT, IPC_SET, or IPC_RMID, the value returned may be the following:

0 Completed successfully.

DIAGNOSTICS

`Errno` may be set to one of the following error code regardless of the value of *cmd*:

EINVAL *Semid* is not a valid semaphore set identifier, or *cmd* is invalid.

If *cmd* is GETVAL, GETPID, GETNCNT, or GETZCNT, `errno` may be set to one of these values:

EINVAL *Semnum* is less than zero or greater than the number of semaphores in the semaphore set identified by *semid*.

EACCES Read permission is denied to the calling process.

If *cmd* is SETVAL, `errno` may be set to one of the following:

EINVAL *Semaphore* is less than zero or greater than the number of semaphores in the semaphore set identified by *semid*.

EACCES Alter permission is denied to the calling process.

ERANGE The value to which the selected semaphore is to be set, *arg.val*, is greater than the system-imposed maximum.

If *cmd* is GETALL, `errno` may be set to one of these values:

EACCES Read permission is denied to the calling process.

EFAULT *Argument.array* points to an illegal address.

If *cmd* is SETALL, `errno` may be set to one of these values:

EACCES Alter permission is denied to the calling process.

ERANGE The value to which one of the semaphores is to be set is greater than the system-imposed maximum.

EFAULT *Argument.array* points to an illegal address.

If *cmd* is IPC_STAT, `errno` may be set to one of these values:

EACCES Read permission is denied to the calling process.

EFAULT *Argument.buf* points to an illegal address.

If *cmd* is IPC_SET, `errno` may be set to one of these values:

EPERM Permission to change the semaphore set attributes is denied to the calling process.

EFAULT *Argument.buf* points to an illegal address.

If *cmd* is IPC_RMID, `errno` may be set to this value:

EPERM Permission to remove the semaphore set is denied to the calling process.

SEE ALSO

`intro`(2), `ipcrm`(1), `ipcs`(1), `semget`(2), `semop`(2).

NAME

semget – get a set of semaphores

SYNOPSIS

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

int   semget (key, nsems, semflg)
key_t key;
int   nsems;
int   semflg;
```

where:

key A user-defined name for the semaphore set

nsems The requested number of semaphores in the semaphore set

semflg A set of flags indicating the requested permission state of the semaphore set, whether a new semaphore set should be created, and whether the semaphore set should be held exclusively

DESCRIPTION

`Semget` returns the semaphore set identifier associated with *key*. This semaphore set identifier may then be used in other semaphore set operations as specified by `semctl` and `semop`. `Semget` can be used to get the semaphore set identifier of an already existing semaphore set or to create a new semaphore set with *nsems* semaphores.

Four options are available:

- Create a private semaphore set.

  In this case, *key* is IPC_PRIVATE.

  A process can create a "private" semaphore set by using the special IPC_PRIVATE key. The system will create a semaphore set identifier that is private to the process. The semaphore set identifier will not be returned to other processes regardless of what key value they specify.

  The newly created semaphore set can be shared among other processes by distributing the semaphore set identifier.

  A process can make multiple semget operations specifying IPC_PRIVATE. The identifiers returned will be unique and the associated semaphore sets will be different.

- Find *key* if already defined.

  In this case, the IPC_CREAT and IPC_EXCL bits of *semflg* are clear and *key* is not IPC_PRIVATE.

  The semaphore set identifier associated with the given key is returned. An error is given if one of the following conditions hold:

  - No semaphore set identifier is associated with *key*.
  - A semaphore set identifier is associated with *key* but the permission rights of the semaphore set do not include those specified by the low-order 9 bits of *semflg*.

  - A semaphore set identifier is associated with *key* but *nsems* is non-zero and the number of semaphores in the semaphore set is less than *nsems*.

- Create only if *key* not already defined.

  In this case, the IPC_CREAT and IPC_EXCL bits of *semflg* are both set and *key* is not IPC_PRIVATE.

  If a semaphore set identifier already exists for *key* an error is returned. Otherwise, a semaphore set identifier and associated semaphore set are created. The semaphore set identifier will be returned to other processes that specify the same key value.

- Find *key* if already defined, otherwise create.

  In this case, the IPC_CREAT bit of *semflg* is set, the IPC_EXCL bit of *semflg* is clear, and *key* is not IPC_PRIVATE.

  If a semaphore set identifier already exists for *key*, this is identical to the second option above). Otherwise, this is identical to the third option above.

If a new semaphore set is created, its attributes are initialized as follows:

- The semaphore set creator's user id (sem_perm.cuid) and the semaphore set's user id (sem_perm.uid) are set to the effective user id of the calling process.
- The semaphore set creator's group id (sem_perm.cgid) and the semaphore set's group id (sem_perm.gid) are set to the effective group id of the calling process.
- The semaphore set's permission rights (in sem_perm.mode) are set to the low-order 9 bits of *semflg*.
- The number of semaphores in the semaphore set (sem_nsems) is set to *nsems*.
- The most recent time a semop operation was performed (sem_otime) is set to the zero value.
- The most recent time the semaphore set attributes were changed (sem_ctime) is set to the current time.
- The semaphore's value is set to zero.
- The process id of the last process to perform an operation on the semaphore is set to zero.
- The number of processes waiting for the semaphore's value to become zero is zero.
- The number of processes waiting for the semaphore's value to increase is zero.
- No processes have a semaphore adjustment value for the semaphore.

ACCESS CONTROL

See the description of the exception conditions EACCESS below.

RETURN VALUE

semid A non-negative integer that identifies the semaphore set associated with *key*.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

If a semaphore set identifier exists for *key*, `errno` may be set to one of these values:

EACCES The permission rights of the semaphore set do not include those specified by the low-order 9 bits of *semflg*.

EINVAL *nsems* is non-zero, and the number of semaphores in the set associated with *key* is less than *nsems*.

EEXIST Both the IPC_CREAT and IPC_EXCL bits of *semflg* are set.

If a semaphore set identifier does not exist for *key*, `errno` may be set to one of these values:

EINVAL *nsems* is either less than or equal to zero or greater than the system-imposed limit.

ENOENT The IPC_CREAT bit of *semflg* is clear.

ENOSPC Creating the new semaphore set would cause the system-imposed limit on the maximum number of allowed semaphore sets system-wide to be exceeded.

SEE ALSO

`intro(2)`, `ipcrm(1)`, `ipcs(1)`, `semctl(2)`, `semop(2)`.

NAME

semop – semaphore operations

SYNOPSIS

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

int      semop (semid, sops, nsops)
int      semid;
struct sembuf * sops;
unsigned nsops;
```

where:

semid A semaphore set identifier

sops An array of semaphore operations to perform

nsops The number of semaphore operation records in *sops*

DESCRIPTION

Semop atomically performs *nsops* semaphore operations on the semaphore set identified by *semid*. The semaphore operation records are located in the array given by *sops*.

A semaphore operation has three components: a semaphore number (*sem_num*), a semaphore operation (*sem_op*), and an operation modifier (*sem_flg*). A semaphore operation is performed by applying the operation, *sem_op*, modified by *sem_flg*, to the semaphore specified by *sem_num* in the semaphore set identified by *semid*. *Sem_op* specifies one of three semaphore operations as follows:

- *sem_op* < 0: **Perform a P operation**

If the semaphore's value is greater than or equal to the absolute value of *sem_op*, the operation is successful. In this case, the absolute value of *sem_op* is subtracted from the semaphore's value, and, if the SEM_UNDO bit of *sem_flg* is set, the absolute value of *sem_op* is added to the calling process's semaphore adjustment value for the semaphore.

If the semaphore's value is less than the absolute value of *sem_op* and the IPC_NOWAIT bit of *sem_flg* is set, semop will return with the error condition EAGAIN.

If the semaphore's value is less than the absolute value of *sem_op* and the IPC_NOWAIT bit of *sem_flg* is clear, semop will suspend the calling process until:

  - the semaphore's value becomes greater than or equal to the absolute value of *sem_op*, in which case, the operation is retried,
  - *semid* is removed from the system, in which case, semop will return with the error condition EIDRM, or
  - the calling process receives a signal that is to be caught, in which case, semop will return with the error condition EINTR.

- *sem_op* > 0: **Perform a V operation**

The value of *sem_op* is added to the semaphore's value and, if the SEM_UNDO bit of *sem_flg* is set, the value of *sem_op* is subtracted from the calling process's semaphore adjustment value for the semaphore.

- *sem_op* == 0: **Wait for zero**

If the semaphore's value is zero, the operation is successful.

If the semaphore's value is non-zero and the IPC_NOWAIT bit of semflg is set, semop will return with the error condition EAGAIN.

If the semaphore's value is non-zero and the IPC_NOWAIT bit of *sem_flg* is clear, semop will suspend the calling process until:

  - the semaphore's value becomes zero or equal to the absolute value of *sem_op*, in which case, the operation is retried,
  - *semid* is removed from the system, in which case, semop will return with the error condition EIDRM, or
  - the calling process receives a signal that is to be caught, in which case, semop will return with the error condition EINTR.

The operation performed by semop is the composition of the individual operations given by *sops*. This composition is subject to the following:

- Semop performs the composite operation atomically. Semop succeeds and changes all subject semaphores only when all individual operations can be performed at a given time.
- The calling process is suspended due to the first individual operation in *sops* that requires suspension.
- When the calling process is suspended, it is registered as waiting for only one semaphore's value to either increase or become zero, that semaphore being the subject semaphore of the individual operation that suspended the process.
- Neither the individual operations nor the composite operation are guaranteed to solve the mutual exclusion livelock problem.

If semop fails, the semaphore set will remain unchanged. Upon successful completion, the calling process is recorded as the last process to perform an operation on each semaphore specified in the array pointed to by *sops* and the current time is recorded as the most recent time a semop operation was performed.

ACCESS CONTROL

Alter access to the semaphore set is required to change the value of a semaphore. Read access is required to wait for a semaphore value to become zero.

RETURN VALUE

0	Completed successfully.
−1	An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EINVAL	*Semid* is not a valid semaphore set identifier.
EINVAL	The number of semaphore sets for which the calling process requests a SEM_UNDO would exceed the system-imposed limit.
EFBIG	*Sem_num* is less than zero or greater than or equal to the number of semaphores in the set associated with *semid* for one or more semaphore operations.
E2BIG	*nsops* is greater than the system-imposed maximum.

EACCES	One of the semaphore operations (*sem_op*) is non-zero and the caller does not have write access.
EACCES	One of the semaphore operations (*sem_op*) is zero and the caller does not have read access.
EFAULT	*sops* is an illegal address.
EAGAIN	Semaphore operations would result in suspension of the caller who has requested IPC_NOWAIT
ENOSPC	The limit on the number of processes requesting a SEM_UNDO would be exceeded.
ERANGE	One of the semaphore operations would cause a semaphore's value to overflow the system-imposed limit.
ERANGE	One of the semaphore operations would cause a process's semaphore adjustment value to overflow the system-imposed limit.
EIDRM	*semid* was removed from the system while the caller was suspended by `semop`.
EINTR	The caller received a signal that was set to be caught while suspended by `semop`.

SEE ALSO

`ipcrm`(1), `ipcs`(1), `intro`(2), `exec`(2), `fork`(2), `semctl`(2), `semget`(2), `exit`(3C).

NAME

semsys – perform a semaphore operation

SYNOPSIS

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

int  semsys  (P1, P2, P3, P4, P5)
int  P1;
int  P2;
int  P3;
int  P4;
int  P5;
```

where:

P1 An integer indicating the type of operation to be performed with a semaphore (0 = SEMCTL, 1 = SEMGET, 2 = SEMOP)

P2 The semaphore set key, if the *P1* operation is SEMGET; otherwise, the semaphore set id

P3 A value that varies based on the following *P1* operations:
SEMCTL The number of some semaphore in the set
SEMGET The number of semaphores in the set
SEMOP A pointer to the semaphore operation structures (`sembuf`)

P4 A value that varies based on the following *P1* operations:
SEMCTL The control command number
SEMGET A flag regulating the type of SEMGET operation and access to the semaphore set
SEMOP The number of semaphore operations to perform

P5 If the operation is SEMGET or SEMOP, *P5* is invalid. In case of SEMCTL, *P5* is a union of `semun` type.

DESCRIPTION

The `semsys` system call performs a semaphore operation (SEMGET, SEMCTL, SEMOP) indicated by the value of *P1*.

ACCESS CONTROL

See the description of the exception condition EACCES in `semget`(2), `semop`(2), and `semctl`(2).

RETURN VALUE

semid A non-negative integer that identifies the semaphore set identifies the semaphore set associated with *key*.

0 `semctl` or `semop` was successful.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

The error codes returned depend on the type of semaphore operations performed and are described in `semctl`(2), `semget`(2), and `semop`(2).

EINVAL *P1* argument is not in the range of 0 through 2.

SEE ALSO

`intro`(2), `semctl`(2), `semget`(2), `semop`(2).

NAME

`send` – send a message from a socket

SYNOPSIS

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

int  send (s, msg, len, user_flags)
int  s;
char * msg;
int  len;
int  user_flags;
```

where:

s	File descriptor of the socket to send message from
msg	Message buffer
len	Length of message (in bytes)
user_flags	Flags to use when sending

DESCRIPTION

`Send` transmits a message to another socket. `Send` can be used only when the socket is connected.

The length of the message is given by the *len* argument. If the message is too long to pass atomically through the underlying protocol, then the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a send. Return values of –1 indicate some locally detected errors.

If no message space is available at the socket to hold the message to be transmitted, then send normally blocks, unless the socket has been placed in non-blocking I/O mode. The `select` call determines when you may send more data.

The *user_flags* parameter may be set to SOF_OOB to send out-of-band data on sockets that support this notion (e.g., SOCK_STREAM).

ACCESS CONTROL

None.

RETURN VALUE

1..*len*	Completed successfully. The call returns the number of characters sent.
–1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	The argument *s* is not an active valid file descriptor.
ENOTSOCK	The argument *s* is not a socket.
EFAULT	An invalid user space address was specified for a parameter.
EMSGSIZE	The socket requires that message be sent atomically, and the size of the message made this impossible.
EAGAIN	The socket is marked non-blocking and the requested operation would block.
EINTR	The send was interrupted by delivery of a signal before any data was delivered.

ENOTCONN Tried to send on unconnected socket.

EOPNOTSUPP The *flags* argument included the MSG_OOB flag applied to a UDP socket.

EPIPE An established connection on a SOCK_STREAM socket was closed by the remote peer.

SEE ALSO

recv(2), socket(2).

NAME

sendmsg – send a message from a socket

SYNOPSIS

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

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

where:

s	File descriptor of socket to send message from
msg	Message header packet
user_flags	Flags to use when sending

DESCRIPTION

The sendmsg call performs the same function as send or sendto except the arguments are repackaged in msghdr. The msghdr structure allows use of the IOV structure [see readv(2) for a description of IOV] for access to non-contiguous buffers.

If the msg_name field of msghdr is null, this call is identical to send. If the msg_name field of msghdr is not null, it identifies the name of the destination for the message and this call is identical to sendto.

ACCESS CONTROL

None.

RETURN VALUE

1..*len*	Completed successfully. The call returns the number of characters sent.
–1	An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EBADF	The argument *s* is not an active valid descriptor.
ENOTSOCK	The argument *s* is not a socket.
EFAULT	An invalid user space address was specified for a parameter.
EMSGSIZE	The socket requires that message be sent atomically, and the size of the message made this impossible.
EAGAIN	The socket is marked non-blocking and the requested operation would block.
ENOTCONN	The socket is unconnected and requires a destination address be specified.
EISCONN	The socket is connected and cannot accept a destination address.
EINTR	The sendmsg() was interrupted by delivery of a signal before any data was delivered.

SEE ALSO

recv(2), socket(2).

NAME

sendto – send a message from a socket

SYNOPSIS

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

int     sendto (s, msg, len, user_flags, to, tolen)
int     s;
char *  msg;
int     len;
int     user_flags;
struct sockaddr *to;
int     tolen;
```

where:

s	File descriptor of socket to send message from
msg	Message buffer
len	Length of message (in bytes)
user_flags	Flags to use when sending
to	Name of destination
tolen	Length of destination name in bytes

DESCRIPTION

This call sends a message, as does send. However, sendto has arguments that specify the destination of the message.

The address of the destination is given by the to argument with *tolen* specifying its size. When used with a connected socket, the *to* and *tolen* arguments are ignored. Other arguments are the same as for send.

ACCESS CONTROL

None.

RETURN VALUE

1..*len*	Completed successfully. The call returns the number of characters sent.
−1	An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EBADF	The argument *s* is not an active valid descriptor.
ENOTSOCK	The argument *s* is not a socket.
EFAULT	An invalid user space address was specified for a parameter.
EMSGSIZE	The socket requires that message be sent atomically, and the size of the message made this impossible.
EAGAIN	The socket is marked non-blocking and the requested operation would block.
EISCONN	Can't sendto with connected socket.
EINTR	The sendto() was interrupted by delivery of a signal before any data was delivered.

SEE ALSO

recv(2), socket(2).

NAME

`setdomainname` – set name of current domain

SYNOPSIS

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

where:

name Name to set for domain

namelen Length of name in bytes

DESCRIPTION

`Setdomainname` sets the domain of the host node to *name*, which has length *namelen*. Only the superuser may use this call; it is normally used at boot time.

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

Domain names are limited to MAXDOMAINNAMELEN characters, which is defined in `<user/param.h>`.

ACCESS CONTROL

Only the superuser can set the domain name.

RETURN VALUE

0 Completed successfully.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EFAULT The *name* parameter gave an invalid address, or the *namelen* parameter specified a length less than zero.

EPERM The caller was not the superuser.

SEE ALSO

`getdomainname`(2), `gethostid`(2), `gethostname`(2).

NAME

setegid – set the effective group id of the current process

SYNOPSIS

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

pid_t setegid(gid_t egid);
```

where:

egid The effective group identifier

DESCRIPTION

If the effective-user-id of the calling process is superuser, effective-group-id is set to *egid*. If the effective-user-id of the calling process is not superuser, but its real-group-id or its effective-group-id or its saved-group-id is equal to *egid*, the effective-group-id is set to *egid*.

ACCESS CONTROL

See the description above.

RETURN VALUE

Upon successful completion, the function setegid returns zero. Otherwise, it returns -1 and sets errno to indicate an error.

DIAGNOSTICS

Under the following conditions, the function setegid fails and sets errno to:

EPERM An attempt was made to set the effective-group-id to a value not permitted by the access control restrictions described above.

EINVAL The supplied value of *egid* was negative or greater than the maximum allowable user id.

SEE ALSO

getuid(2), geteuid(2), getgid(2), setuid(2), setgid(2), setregid(2), setreuid(2).

NAME

`seteuid` – set the effective user id of the current process

SYNOPSIS

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

pid_t seteuid(gid_t euid);
```

where:

euid An effective user identifier

DESCRIPTION

If the effective-user-id of the calling process is superuser, effective-user-id is set to *euid*.

If the effective-user-id of the calling process is not superuser, but its real-user-id or its effective-user-id or its saved-user-id is equal to *euid*, the effective-user-id is set to *euid*.

ACCESS CONTROL

See the description above.

RETURN VALUE

Upon successful completion, the function `seteuid` returns zero. Otherwise, it returns −1 and sets `errno` to indicate an error.

DIAGNOSTICS

Under the following conditions, the function `seteuid` fails and sets `errno` to:

EPERM An attempt was made to set the effective-user-id to a value not permitted by the access control restrictions described above.

EINVAL The supplied value of *euid* was negative or greater than the maximum allowable user id.

SEE ALSO

`getuid(2)`, `geteuid(2)`, `getgid(2)`, `setuid(2)`, `setgid(2)`, `setregid(2)`, `setreuid(2)`.

NAME

setgid – set the real-, effective-, and saved-group-ids

SYNOPSIS

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

int  setgid (gid)
gid_t  gid;
```

where:

gid The value to which the calling process's group-ids are to be set

DESCRIPTION

Setgid sets the real-group-id, effective-group-id, and saved-group-id of the calling process to *gid*, subject to the access control constraints described below.

The value of *gid* must always be non-negative and less than MAXUID.

ACCESS CONTROL

If the effective-user-id of the calling process is superuser the real-group-id, effective-group-id, and saved_group_id values are all set to *gid*.

If the effective-user-id of the calling process is not superuser, but its real-group-id or its saved_group_id is equal to *gid*, the effective-group-id is set to *gid*. The real-group-id and saved_group_id are unchanged.

RETURN VALUE

0 Successful completion.

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EPERM An attempt was made to set the effective-group-id to a value not permitted by the access control restrictions described above.

EINVAL The supplied value of *gid* was negative or greater than MAXUID.

SEE ALSO

getegid(2), geteuid(2), getgid(2), getuid(2), setregid(2), setreuid(2), setuid(2).

NAME

`sethostid` – set unique identifier of current host

SYNOPSIS

```
int    sethostid (new_hostid)
long   new_hostid;
```

where:

new_hostid   Hostid to set

DESCRIPTION

`Sethostid` establishes an identifier for the current node, which is intended to be unique among all UNIX systems in existence. This is normally a DARPA Internet address for the local machine. Only the superuser may use this call; it is normally performed at boot time.

ACCESS CONTROL

The effective user id of the calling process must be superuser.

RETURN VALUE

0   Completed successfully.

–1   An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to the following error code:

EPERM   Caller must be superuser.

SEE ALSO

`hostid`(1), `getdomainname`(2), `gethostid`(2), `gethostname`(2).

NAME

`sethostname` – set name of current host

SYNOPSIS

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

where:

name Name to set for host

namelen Length of name in bytes

DESCRIPTION

`Sethostname` sets the name of the host node to *name*, which has length *namelen*. Only the superuser may use this call; it is normally used at boot time.

Hostnames are limited to MAXHOSTNAMELEN characters, which is defined in `<sys/param.h>`.

ACCESS CONTROL

Only the superuser can set the hostname.

RETURN VALUE

0 Completed successfully.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EFAULT The *name* parameter gave an invalid address, or the *namelen* parameter specified a length less than zero.

EPERM The caller was not the superuser.

SEE ALSO

`getdomainname`(2), `gethostid`(2), `gethostname`(2). `uname`(1), `uname`(2)

NOTES

This system call also modifies the node name that is contained in the system's utsname structure. Subsequent calls to uname -n will return this new node name.

NAME

`setpgid` – set process group ID for job control

SYNOPSIS

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

int setpgid (pid, pgid)
pid_t pid, pgid;
```

where:

pid The process id of the process whose process group id is to be changed.

pgid The new process group id.

DESCRIPTION

The `setpgid()` function is used to either join an existing process group or create a new process group within the session of the calling process. The process group ID of a session leader shall not change. Upon successful completion, the process group ID of the process with a process ID that matches *pid* shall be set to *pgid*. As a special case, if *pid* is zero, the process ID of the calling process shall be used. Also, if *pgid* is zero, the process ID of the indicated process shall be used.

RETURN VALUE

0 Successful completion.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

If any of the following conditions occur, the `setpgid()` function shall return −1 and set `errno` to the corresponding value:

EACCES The value of the *pid* argument matches the process ID of a child process of the calling process and the child process has successfully executed one of the `exec()` functions.

EINVAL The value of the *pgid* argument is less than zero or is not a value supported by the implementation.

ENOSYS The `setpgid()` function is not supported by this implementation.

EPERM The process indicated by the *pid* argument is a session leader.

The value of the *pid* argument is valid but matches the process ID of a child process of the calling process and the child process is not in the same session as the calling process.

The value of the *pgid* argument does not match the process ID of the process indicated by the *pid* argument and there is no process with a process group ID that matches the value of the *pgid* argument in the same session as the calling process.

ESRCH The value of the *pid* argument does not match the process ID of the calling process or of a child process of the calling process.

SEE ALSO

`exec`(2), `getpgrp`(2), `setsid`(2), `tcsetpgrp`(3C).

COPYRIGHTS

Department. To purchase IEEE Standards, call 800/678-IEEE.

In the event of a discrepancy between the electronic and the original printed version, the original version takes precedence.

STANDARDS

The `setpgid()` always behaves as if _POSIX_JOB_CONTROL were defined, regardless of whether or not it is defined.

NAME

setpgrp – set process-group-id

SYNOPSIS

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

gid_t  setpgrp ()
```

DESCRIPTION

The setpgrp system call sets the process-group-id of the calling process to the process-id of the calling process.

ACCESS CONTROL

No access checking is performed.

RETURN VALUE

process-group-id   The new value of the calling process's process-group-id.

DIAGNOSTICS

None.

SEE ALSO

getpgrp(2), getpgrp2(2), setpgrp2(2).

STANDARDS

When using m88kbcs as the Software Development Environment target, the setpgrp function will be emulated using BCS system calls. Since this is an emulation requiring several BCS system calls, a slight performance degradation may be noticed in comparison to using berk_signal in /lib/libc.a.

The only way for a session to allocate a controlling terminal is for the session leader (which must not already have a controlling terminal) to open a terminal device that is not already associated with any session, without using the O_NOCTTY option to open. The effect is that the processes in a session may have at most one controlling terminal, and a terminal may have at most one controlling process, which must be a session leader. When the controlling process terminates, an automatic vhangup occurs on the terminal and its terminal characteristics are reset.

NAME

`setpgrp2` – set process-group-id

SYNOPSIS

```
#include &lt;sys/types.h&gt;

int  setpgrp2 (pid, pgrp)
pid_t  pid;
gid_t  pgrp;
```

where:

pid The process-id of the process whose process-group-id is to be changed. A value of zero denotes the calling process, not pid 0.

pgrp The value to which the target process's process-group-id is to be set

DESCRIPTION

If the access control requirements described below are met, `setpgrp2` sets the process-group-id of the process specified by *pid* to the value specified by *pgrp*. The value of *pgrp* is not required to be the process-id of an existing process; hence a process group with no group leader can be established.

ACCESS CONTROL

The access control requirements of `setpgrp2` can be met in one of three ways: 1) the caller has effective-user-id of superuser, or 2) the target process is a descendant of the caller in the process tree, or 3) the target process has the same effective-user-id as the caller.

RETURN VALUE

0 Successful completion.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

ESRCH The process specified by *pid* does not exist.

EPERM None of the three conditions described in the ACCESS CONTROL section above is met.

SEE ALSO

`getpgrp(2)`, `getpgrp2(2)`, `setpgrp(2)`.

STANDARDS

When using `m88kbcs` as the Software Development Environment target, the `setpgrp2` function will be a restricted emulation of Berkeley semantics. This emulation only allows a process to join a process group already in use inside its session or to create a new process group whose process group ID is equal to its process ID.

NAME

`setpriority` – set process scheduling priority

SYNOPSIS

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

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

where:

which — How the argument *who* is to be interpreted in identifying one or more processes whose priorities will be set: PRIO_PROCESS, PRIO_PGRP, or PRIO_USER

who — Identifier of one or more processes whose priorities will be set: a process ID, a process group ID, or user ID, depending on the value of *which*

prio — The new priority value

DESCRIPTION

One or more processes are identified by the combination of the arguments *which* and *who*. If *which* is PRIO_PROCESS, *who* is interpreted as a process ID and a single process is identified. If *which* is PRIO_PGRP, *who* is interpreted as a process group ID, and all processes that are members of that group are identified. If *which* is PRIO_USER, *who* is interpreted as a user ID, and all processes with an effective user id of *who* are identified. A *who* value of 0 is interpreted as the calling process's process ID, process group ID, and effective-user-id, respectively, for the three cases listed. For example, all processes in the calling process's process group may be identified with *which* set to PRIO_PGRP and *who* set to zero.

The `setpriority` call sets the priorities of all the identified processes to *prio*, subject to the access control constraints described below. The access checks are applied to each process in the identified set. If one or more processes fail the checks, `setpriority` still changes the priority of those processes that pass the checks, but the error return value will be given.

ACCESS CONTROL

In order to set a process's priority to a larger numerical value (less favorable scheduling) or leave it unchanged, the calling process must have an effective-user-id that is 0 or that matches the target process's effective-user-id.

In order to set a process's priority to a smaller numerical value, the calling process must have an effective-user-id that is 0.

RETURN VALUE

0 — Successful completion.

–1 — An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

ESRCH — Using the *which* and *who* values specified, no processes were located at all, or if any processes were located, none passed the access checks.

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

EACCES One or more (but not all) of the processes in the identified set did not pass the access checks described above.

SEE ALSO

fork(2), nice(2).

NAME

`setpsr` – set the processor status register

SYNOPSIS

```
#include <sys/m88kbcs.h>

unsigned int  setpsr (new_psr)
unsigned int  new_psr;
```

where:

new_psr The bits to be set or cleared in the calling process's psr

DESCRIPTION

This system call sets several bits in the Processor Status Register of the calling process. These bits control certain aspects of the execution of the process. The bits that may be set are the SER, C, BO, and MXM bits.

Setting the SER bit turns on serial mode. Clearing this bit allows concurrent operation. Setting the C bit sets the carry bit to one. Clearing this bit sets the carry to zero. Setting the MXM bit disables misaligned access exceptions. Clearing this bit enables misaligned access exceptions; in this mode a misaligned access causes the system to deliver a SIGBUS signal to the process. Setting the BO bit causes the current byte ordering to be Litte- Endian; clearing this bit causes the current byte ordering to be Big-Endian. Regardless of the setting of the BO bit, all interfaces to or from the system are always in Big-Endian order. All other bits are ignored.

This call returns the previous setting of the Processor Status Register.

ACCESS CONTROL

No access checking is performed.

RETURN VALUE

processor status register
The processor status register of the calling process.

DIAGNOSTICS

None.

SEE ALSO

`getpsr`(2).

NAME

`setregid` – set the real-, effective-, and saved-group-ids

SYNOPSIS

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

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

where:

rgid The value to which the real-group-id should be set

egid The value to which the effective-group-id should be set

DESCRIPTION

The real-group-id and effective-group-id's of the calling process are set according to the arguments. If *rgid* or *egid* is −1, the current value of the real-group-id is used. If the caller is not the superuser, he may only set the effective-group-id to the real-group-id or the saved-group-id. Only the superuser may make other changes. If after changing the real- and effective-group-id's, the calling process's effective-group-id no longer matches either its real- or saved-group-id's, its saved-group-id is set to the value of its effective-group-id. If the real-group-id is changed and the calling process's group list is not empty, the old real-group-id is deleted from the group list and the new real-group-id is added.

Note that since the effective-group-id is implicitly a part of the group list, if this call changes the effective-group-id it also changes the group list.

ACCESS CONTROL

If the calling process has effective-user-id of superuser, setting of the real- and effective-user-ids is not restricted.

Otherwise, the effective-user-id may be set only to its current value or to the current value of the real-user-id or to the saved-user-id value. The real-user-id may be set only to its current value.

RETURN VALUE

0 Successful completion.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EPERM The above specified access check failed.

EINVAL The supplied value of *rgid* or *egid* was less than −1 or greater than MAXUID.

SEE ALSO

`getuid(2)`, `geteuid(2)`, `getgid(2)`, `getegid(2)`, `setuid(2)`, `setgid(2)`, `setreuid(2)`.

NAME

`setreuid` – set the real-, effective-, and saved-user-ids

SYNOPSIS

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

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

where:

ruid The value to which the real-user-id should be set

euid The value to which the effective-user-id should be set

DESCRIPTION

The real-user-id and effective-user-id's of the calling process are set according to the arguments. If *ruid* or *euid* is –1, the current value of the real-user-id is used. If the caller is not the superuser, he may only set the effective-user-id to the real-user-id or the saved_user_id. Only the superuser may make other changes. If after changing the real- and effective-user-id's, the calling process's effective-user-id no longer matches either its real- or saved-user-id's, its saved-user-id is set to the value of its effective-user-id.

ACCESS CONTROL

If the calling process has effective-user-id of superuser, setting of the real- and effective-user-ids is not restricted.

Otherwise, the effective-user-id may be set only to its current value or to the current value of the real-user-id or to the saved-user-id value. The real-user-id may be set only to its current value.

RETURN VALUE

0 Successful completion.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EPERM The above specified access check failed.

EINVAL The supplied value of *ruid* or *euid* was less than –1 or greater than MAX-UID.

SEE ALSO

`getuid`(2), `geteuid`(2), `getgid`(2), `getegid`(2), `setuid`(2), `setgid`(2).

NAME

setsid – create session and set process group ID

SYNOPSIS

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

pid_t  setsid ()
```

DESCRIPTION

If the calling process is not a process group leader, the setsid() function shall create a new session. The calling process shall be the session leader of this new session, shall be the process group leader of a new process group, and shall have no controlling terminal. The process group ID of the calling process shall be set equal to the process ID of the calling process. The calling process shall be the only process in the new process group and the only process in the new session.

RETURN VALUE

Upon successful completion, the setsid() function returns the value of the process group ID of the calling process.

DIAGNOSTICS

If any of the following conditions occur, the setsid() function shall return -1 and set errno to the corresponding value:

EPERM — The calling process is already a process group leader or the process group ID of a process other than the calling process matches the process ID of the calling process.

SEE ALSO

exec(2), _exit(2), fork(2), getpid(2), kill(2) setpgid(2), sigaction(2).

COPYRIGHTS

STANDARDS

The only way for a session to allocate a controlling terminal is for the session leader (which must not already have a controlling terminal) to open a terminal device that is not already associated with any session, without using the O_NOCTTY option to open(). The effect is that the processes in a session may have at most one controlling terminal, and a terminal may have at most one controlling process, which must be a session leader. When the controlling process terminates, an automatic vhangup() occurs on the terminal and its terminal characteristics are reset.

NAME

`setsockopt` – set options on sockets

SYNOPSIS

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

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

where:

s	File descriptor of socket to set options on
level	Level in socket that the options apply to (e.g., socket level, implementing protocol level)
optname	Name of options to set
optval	Value associated with option
optlen	Length of option to set (bytes)

DESCRIPTION

The `setsockopt` call sets options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost socket level.

When setting socket options, the caller must specify the level at which the option resides and the name of the option. To manipulate options at the socket level, *level* is specified as SOL_SOCKET. To manipulate options at any other level, the protocol number of the appropriate protocol controlling the option is supplied. See documentation for the domain being used.

The parameters *optval* and *optlen* supply option values for `setsockopt`. If no option value is to be supplied, *optlen* must be supplied as 0 and *optval* may be undefined.

Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file `<sys/socket.h>` contains definitions for socket level options; see `socket`(2). Options at other protocol levels vary in format and name; consult the related domain documentation.

Socket Level Options

This is a list of the options recognized at the socket level:

SO_DEBUG	Toggles debugging in the underlying protocol modules. *optval* is a pointer to on/off flag *int*.
SO_REUSEADDR	Toggles the indication that the rules used in validating addresses supplied in a `bind`(2) call shall allow reuse of local addresses. *optval* is a pointer to on/off flag *int*.
SO_KEEPALIVE	Toggles the periodic transmission of messages on a connected socket. Should the connected peer fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a SIGPIPE signal. *optval* is a pointer to on/off flag *int*.
SO_DONTROUTE	Toggles the indication that outgoing messages shall bypass the standard routing facilities. Instead, messages are directed to

	the appropriate network interface according to the network portion of the destination address. *optval* is a pointer to on/off flag *int*.
SO_LINGER	Controls the action taken when unsent messages are queued on the socket and a `close(2)` is performed. If linger is set, the system will block the process on `close(2)` until all the data is sent or until the linger timeout expires. A linger timeout of zero will cause the system to process the close in a manner that allows the process to continue as quickly as possible. If linger is reset, the system will block the process on `close(2)` untill all the data is sent or the system detects that the connection is no longer viable. *optval* is a pointer to `struct linger`.
SO_BROADCAST	Toggles permission to send broadcast datagrams on the socket. *optval* is a pointer to on/off flag *int*.
SO_OOBINLINE	With protocols that support out-of-band data, the option toggles the request that out-of-band data be placed in the normal data input queue as received; it will then be accessible with `recv(2)` or `read(2)` calls without the MSG_OOB flag. *optval* is a pointer to on/off flag *int*.
SO_SNDBUF	Adjusts the normal buffer sizes allocate for output buffers. *optval* is a pointer to *int* containing the size of send buffer.
SO_RCVBUF	Adjust the normal buffer sizes allocated for input buffers. *optval* is a pointer to *int* containing the size of receive buffer.
SO_TYPE	Used only with `getsockopt(2)` to return the type of the socket. *optval* is a pointer to *int* containing the socket type.
SO_ERROR	Used only with `getsockopt(2)`. It returns any pending error on the socket and clears the error status. *optval* is a pointer to *int* containing `errno`.

ACCESS CONTROL

Consult domain documentation for any specific restrictions imposed by the domain. SOL_SOCKET has no restrictions.

RETURN VALUE

0	Completed successfully.
−1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF	The argument *s* is not an active valid descriptor.
ENOTSOCK	The argument *s* is a file, not a socket.
ENOPROTOOPT	The option is unknown.
EFAULT	The options are not in a valid part of the process address space.
EINVAL	Invalid argument.
ENOBUFS	No internal buffers available.
EOPNOTSUPP	The option is unsupported.
EISCONN	The option is invalid while the socket is in the connected state.

EACCES Caller has inadequate privileges to set the option. Socket privilege is based on the euid of the process when the socket was created.

SEE ALSO

getsockopt(2), socket(2), inet(3N), inet(6F), unix_ipc(6F).

NAME

settimeofday – set date and time

SYNOPSIS

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

int    settimeofday (time_value, time_zone)
struct timeval * time_value;
struct timezone * time_zone;
```

where:

time_value Address of an initialized structure giving the new current time

time_zone NULL or address of an initialized structure giving the new time zone

DESCRIPTION

Settimeofday sets the system's notion of the current Greenwich time and the current time zone to the values contained in the structures at the locations specified by *time_value* and *time_zone*.

When the time is successfully changed, a log of the change is sent to the error logger device.

If *time_zone* is NULL, the current time zone is not changed.

The interpretation of the time value and time zone structures are discussed in gettimeofday.

Although not enforced, it is unusual if the time zone correction, *time_zone.tz_minuteswest*, is not divisible by 60 minutes.

Setting the system clock may interfere with other timing functions.

ACCESS CONTROL

Only the superuser may set the time of day.

RETURN VALUE

0 Completed successfully.

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EFAULT An argument address referenced invalid memory.

EPERM Permission to set the time is denied to the calling process.

SEE ALSO

date(1), gettimeofday(2), ctime(3C).

NAME

`setuid` – set the real-, effective-, and saved-user-ids

SYNOPSIS

```
#include <unistd.h>

int  setuid (uid)
uid_t  uid;
```

where:

uid The value to which the calling process's real-, effective-, and saved-user-ids are to be set

DESCRIPTION

`Setuid` sets the real-user-id, effective-user-id, and saved-user-id of the calling process to *uid*, subject to the access control constraints described below.

The value of *uid* must always be non-negative and less than or equal to MAXUID.

ACCESS CONTROL

If the effective-user-id of the calling process is superuser the real-user-id, effective-user-id, and saved_user_id values are all set to *uid*.

If the effective-user-id of the calling process is not superuser, but its real-user-id or its saved_user_id is equal to *uid*, the effective-user-id is set to *uid*. The real-user-id and saved_user_id are unchanged.

RETURN VALUE

0 Successful completion.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EPERM An attempt was made to set the effective-user-id to a value not permitted by the access control restrictions described above.

EINVAL The supplied value of *uid* was negative or greater than MAXUID.

SEE ALSO

`getegid`(2), `geteuid`(2), `getgid`(2), `getuid`(2), `setgid`(2), `setregid`(2), `setreuid`(2).

NAME

`shmat` – attach a shared memory segment

SYNOPSIS

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

void * shmat (shmid, shmaddr, shmflg)
int shmid;
void * shmaddr;
int shmflg;
```

where:

shmid The shared memory identifier of the shared segment to attach

shmaddr The byte address at which to attach the shared segment (may be defaulted to a system-selected value or rounded to a system-specified address boundary)

shmflg SHM_RND or SHM_RDONLY, an option flag used to select between the various options for *shmaddr* and to choose read-only or read-write access to the shared memory segment

DESCRIPTION

`Shmat` attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process. The segment is attached at the byte address specified by the caller as detailed below, for either read-write access or read-only access. The length of the shared memory segment is taken from the shared memory descriptor associated with *shmid*; i.e., by the value of the shm_segsz field. There is no way to attach only a portion of a shared memory segment.

The address where the segment is attached is returned upon successful completion of the call. This address may be specified in one of three ways:

- Explicitly without rounding.
  If *shmaddr* is non-zero, and *shmflg* & SHM_RND is false, the segment is attached at *shmaddr*. *shmaddr* must be a multiple of the page size; otherwise, an error is returned.
- Explicitly with rounding.
  If *shmaddr* is non-zero, and *shmflg* & SHM_RND is true, the segment is attached at the address obtained by rounding down *shmaddr* to a multiple of SHMLBA, specifically, to (*shmaddr* - (*shmaddr* modulo SHMLBA))
- By default.
  If *shmaddr* is zero, the segment is attached at the first convenient address as selected by the system. NOTE: "first convenient" address means the value is implementation dependent, and may change from release to release. The value is arbitrary and the user should not depend on how the address is selected.

The segment is attached with read-only access if (*shmflg* & SHM_RDONLY) evaluates to true, otherwise it is attached for reading and writing.

Upon successful completion, this call changes the following fields in the shared memory data structure associated with the shared segment:

`shm_lpid` Changed to equal the process identifier of the calling process.

`shm_atime` Changed to equal the current time.

`shm_nattach` Incremented by 1.

There is a per-process limit on the number of shared segments a process may have attached simultaneously. If the process is currently at this system-imposed maximum, the attach operation will not be performed. This limit is the same for ALL processes regardless of process identifier (i.e., this limit does apply to processes whose effective user id is the superuser). This limit is specified by the SHMSEG configuration variable.

A `fork` operation is an implicit attach operation, since a new process inherits all attached shared memory segments from its parent. This implicit attach alters only the `shm_nattach` field as described above for an explicit attach; the `shm_atime` and `shm_lpid` fields are not changed by this implicit attach. Note this implicit attach applies to all attached shared memory segments. This includes IPC_PRIVATE segments, and also segments that have been the target of the IPC_RMID operation of `shmctl`, i.e., have been "deleted" but still exist because their attach account (`shm_nattach`) has not become zero. This exception is the only way such a "deleted" shared memory segment can be attached.

`Shmat` will fail and not attach the shared memory segment if an error occurs.

ACCESS CONTROL

The calling process must have read permission to the shared segment as defined in the shm_perm field of the associated shared memory data structure to attach for read-only access, and read and write permission to attach for read-write access.

RETURN VALUE

address The `shmat` operation was successful; the value returned is the starting byte address of the newly attached shared memory segment.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EINVAL *shmid* is not a valid shared memory identifier.

EINVAL The rounding option was used; i.e., SHM_RND evaluates to true and *shmaddr* is not equal to zero, but the value of (*shmaddr* - (*shmaddr* modulo SHMLBA)) is an invalid address.

EINVAL *shmaddr* was given explicitly; i.e., *shmaddr* is not equal to zero and SHM_RND evaluates to false, but the value of *shmaddr* is not a multiple of the page size, or is an invalid address.

EACCES Operation permission is denied to the calling process.

ENOMEM Either the kernel or the user data space is insufficient to accommodate the attach request. This error may not recur on subsequent calls, if other operations free the needed space.

EMFILE The number of shared memory segments attached to the calling process would exceed the system-imposed limit.

EAGAIN The system-imposed limit on space locked into physical memory would be exceeded; the MCL_FUTURE memory locking option is in effect for the calling process (see `memcntl`(2)).

SEE ALSO

ipcrm(1), ipcs(1), intro(2), exec(2), _exit(2), fork(2), memcntl(2), shmctl(2), shmget(2), exit(3C).

NAME

`shmctl` – shared memory control operations

SYNOPSIS

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

int  shmctl (shmid, cmd, buf)
int  shmid;
int  cmd;
struct shmid_ds * buf;
```

where:

shmid The shared memory identifier of the shared area to be operated on

cmd The specific shared memory operation (IPC_STAT, IPC_SET, or IPC_RMID)

buf The address of a shared memory structure to be used in the operation (used only if command is IPC_STAT or IPC_SET)

DESCRIPTION

The `shmctl` system call obtains or modifies information on shared memory segments previously defined by `shmget`. The `shmctl` call also allows for the destruction of shared memory segments. The action performed by `shmctl` is determined by the value of the *cmd* parameter as follows:

IPC_STAT Get status information. Returns the current value of each member of the shared memory data structure (*shmid_ds*) associated with the shared memory segment specified by *shmid* into the buffer pointed to by *buf*. This command does not change the values of any of the fields in the shared memory data structure. If an error occurs, the contents of the buffer pointed to by *buf* are undefined.

IPC_SET Set status information. Set the current value of each member of the shared memory data structure (*shmid_ds*) associated with the shared memory segment specified by *shmid* to the corresponding value in the buffer pointed to by *buf*. Only the following fields may be set:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode /* only low 9 bits */
```

In addition to setting the above fields, this command causes the `shm_ctime` field to be set to the current time. If an error occurs, no changes are made to any of the fields in the shared memory descriptor.

IPC_RMID Remove shared memory identifier. This "deletes" the shared memory identifier specified by *shmid* from the system. This command has "delete on last detach" semantics. The shared area segment is not actually destroyed until all processes that currently have the area attached detach from it via the `shmdt` operation. However, no other processes may attach to the shared area via the `shmat` call. (Note, however, that implicit attach operations that occur as part of a `fork` operation may still occur; see `shmat`.) Once this command is performed on a *shmid*, the only operations that may be performed on that *shmid* are the IPC_STAT command of `shmctl` and the `shmdt` operation. All other operations act as though *shmid* is invalid; that is, return the EINVAL status code. A shared segment that has had this operation performed on it will have the SHM_DEST bit set in the `shm_perm.mode` field of its shared memory data structure. (This information is returned by the IPC_STAT command of `shmctl`.) This command updates the `shm_ctime` field in the shared segment's data structure to the current time. If an error occurs, the shared segment is not deleted and no changes are made to its shared memory data structure.

Note that none of the commands require that the caller have the shared segment attached.

ACCESS CONTROL

The access required to the shared memory segment denoted by *shmid* depends on the value of *cmd*, as specified below.

IPC_STAT Get status information. The effective user id of the calling process must equal the superuser; or the calling process must have read access (SHM_R) as determined by the mode bits in the `ipc_perm` structure defined by the `shm_perm` field of the shared memory data structure associated with *shmid*.

IPC_SET Set status information. This *cmd* can be executed by any process with effective user id equal to either that of superuser; or to the value of either `shm_perm.uid` or `shm_perm.cuid` in the `ipc_perm` structure defined by the `shm_perm` field of the shared memory data structure associated with *shmid*.

IPC_RMID Remove shared memory identifier. This *cmd* can be executed by any process with effective user id equal to either that of superuser; or to the value of either `shm_perm.uid` or `shm_perm.cuid` in the `ipc_perm` structure defined by the `shm_perm` field of the shared memory data structure associated with *shmid*.

RETURN VALUE

0 The `shmctl` operation was successful.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes for any value of *cmd*:

EINVAL *Shmid* is not a valid shared memory identifier; or *cmd* is not a valid command.

The IPC_STAT command may return the following errors:

EFAULT *Buf* points to an illegal address.

EACCES Read access is denied.

The IPC_SET command may return the following errors:

EFAULT *Buf* points to an illegal address.

EPERM Access is denied; that is, effective user id of the calling process is not superuser and does not match the `shm_perm.uid` or `shm_perm.cuid` fields of the shared memory descriptor associated with *shmid*.

The IPC_RMID command may return the following error:

EPERM Access is denied; that is, effective user id of the calling process is not superuser and does not match the `shm_perm.uid` or `shm_perm.cuid` fields of the shared memory descriptor associated with *shmid*.

SEE ALSO

`intro`(2), `ipcrm`(1), `ipcs`(1), `shmget`(2).

NAME

shmdt – detach a shared memory segment

SYNOPSIS

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

int  shmdt (shmaddr)
void * shmaddr;
```

where:

shmaddr The byte address of the attached shared memory segment to be detached. This must equal the value returned by shmat when the shared segment was attached.

DESCRIPTION

Shmdt detaches the shared memory segment located at the address specified by *shmaddr* from the calling process's data segment.

Upon successful completion, this call changes the following fields in the shared memory data structure associated with the shared segment:

shm_lpid Changed to equal the process id of the calling process.

shm_dtime Changed to equal the current time.

shm_nattach Decremented by 1.

Detaching a shared memory segment makes it no longer available to the calling process. Other users who still have the shared memory segment attached are not affected. However, the calling process may not be able to re-attach to the segment. This will be the case if a remove operation has been performed on the shared memory segment (see the IPC_RMID operation of shmctl).

Calls to either exec or exit cause implicit detach operations on all shared segments that a process has attached. These implicit detach operations change only the shm_nattach field of the shared memory data structure as described above for explicit detach calls. The shm_lpid and shm_dtime fields remain unchanged by these implicit detach operations.

Shmdt will fail and not detach the shared memory segment if an error occurs.

ACCESS CONTROL

No access permission is required to detach a shared memory segment.

RETURN VALUE

0 The detach operation was successful.

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to the following error code:

EINVAL *Shmaddr* is not the starting address of any shared memory segment currently attached to the calling process.

SEE ALSO

intro(2), exec(2), _exit(2), fork(2), ipcrm(1), ipcs(1), shmctl(2), shmget(2), exit(3C).

NAME

shmget – get shared memory segment

SYNOPSIS

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

int    shmget (key, size, shmflg)
key_t  key;
int    size;
int    shmflg;
```

where:

key Key identifying shared memory segment

size Size in bytes of the shared memory segment

shmflg Access and option *shmflgs*. Valid *shmflg* bits that may be set are IPC_CREAT and IPC_EXCL. The low-order 9 bits are the standard access bits.

DESCRIPTION

The shmget system call returns the shared memory identifier associated with *key*. This shared memory identifier may then be used in other shared memory operations as specified by shmat, shmctl, and shmdt. Shmget can be used to get the shared memory identifier of an already existing shared memory segment, or to create a new shared memory segment.

The *size* parameter is used in one of two ways, depending on whether shmget creates a new shared memory segment. When shmget is used to create a new shared memory segment, *size* specifies the number of bytes to make the new shared memory segment. In this case *size* must be greater than or equal to a system-imposed minimum size and less than or equal to a system-imposed maximum size.

When shmget is used to find the shared memory identifier of an existing shared memory segment, *size* is used to ensure any such existing shared memory segment is at least as large as *size*. This guarantees that when the shared memory segment is attached, all references to the shared area whose offsets relative to the start of the shared area are between 0 and *size*−1, inclusive, are valid references to the shared memory segment. If *size* is greater than the value of the existing shared memory segment, an error is returned. If size is less than or equal to the value of the existing shared segment, the shared memory identifier is returned. This is true even if the value of *size* used is less than the system-imposed minimum for creating shared memory segments. In particular, a *size* of 0 can always be specified; this is guaranteed to be less than the actual size of the shared memory segment and therefore passes this test.

Using the IPC_CREAT and IPC_EXCL *shmflgs* in *shmflg* along with the special key value IPC_PRIVATE, four options are available:

- Create a private segment.

  In this case, *key* = IPC_PRIVATE. A process can create a "private" shared memory identifier by using the special IPC_PRIVATE key. This results in the system creating a shared memory identifier that is private to the process. This shared memory id will not be returned to other processes regardless of what key value they specify. Note it is really the key that is "private". The shared memory identifier that is returned is not private; other processes may use this shared memory identifier in other shared memory calls. Thus, the shared memory segment itself is not necessarily private and accessible only to the calling routine. (For example, the process could pass the shared memory identifier to another process via an interprocess message. Even if the process does not do this, the segment is still accessible to any child processes created, since a `fork` operation does an implicit `attach` operation; see `shmat`). A process can make multiple `shmget` calls specifying the IPC_PRIVATE key; the shared memory identifiers returned will be unique and the shared segments associated will be different. Since this call always creates a shared segment, *size* must always be set to the size of the desired segment. If an error occurs, no shared memory segment is created and an error is returned.

- Find *key* if already defined.

  In this case, neither the IPC_CREAT nor the IPC_EXCL *shmflg* bits are set in *shmflg*, and *key* != IPC_PRIVATE. The shared memory identifier associated with the given key is returned. If none exists, or if one exists but the size of the associated segment is less than *size*, an error is returned.

- Find *key* if already defined, otherwise create.

  In this case, the IPC_CREAT *shmflg* bit is set, the IPC_EXCL *shmflg* bit in *shmflg* is ignored, and *key* != IPC_PRIVATE. If a shared memory identifier already exists for *key* and the size of the associated shared memory segment is greater than or equal to *size* (note this will be the case if *size* = 0), the shared memory identifier is returned. If a shared memory identifier already exists for *key* but the size of the associated shared memory segment is less than *size*, an error is returned. If there is no shared memory identifier corresponding to *key*, a shared memory segment and an associated shared memory identifier are created with the specified *key* and *size*. Any errors cause an error to be returned and do not cause a shared memory segment to be created.

- Create only if *key* not currently defined.

  In this case, the IPC_CREAT and IPC_EXCL *shmflg*s are both set in *shmflg*, and *key* != IPC_PRIVATE. If a shared memory identifier already exists for *key*, an error is returned; otherwise, a shared memory segment and associated shared memory identifier are created with the specified *key* and *size*. Since this call attempts to create a shared segment, *size* must always be set to the size of the desired segment.

If a shared memory segment is created, the shared memory data structure associated with the new shared memory identifier is initialized as follows:

- `shm_perm.uid` and `shm_perm.cuid` – set to the effective user id of the calling process.
- `shm_perm.gid` and `shm_perm.cgid` – set to the effective group id of the calling process.
- `shm_perm.mode` – the low-order 9 bits are set to the low-order 9 bits of *shmflg*. Note these bits determine the access to the shared memory segment in the standard way: 3 bits for owner, 3 bits for group, 3 bits for other.
- `shm_ptbl` – set to an implementation-dependent value.

- shm_segsz – set to *size*.
- shm_lpid – set to 0.
- shm_cpid – set to the process id of the calling process.
- shm_nattch – set to 0.
- shm_cnattch – set to 0.
- shm_atime – set to 0.
- shm_dtime – set to 0.
- shm_ctime – set to the current time.

There is a system-imposed maximum on the number of shared memory segments (and therefore shared memory identifiers) that may exist simultaneously. Calls to shmget will fail if they require a new shared memory segment to be created and the system is already at this limit.

In general, applications wishing to share a memory segment must agree on a key in some fashion beforehand. One system-defined mechanism for doing this is the ftok facility, which takes a filename and returns a process-specific key based on that filename. See the ftok description in stdipc(3C).

Although no access permission is required to do a shmget operation, a consistency check is made on the access permissions specified in the lower 9 bits of *shmflg*. For any of the options that return the shared memory identifier of an already existing shared memory segment, a check is made that all mode bits set by the caller in *shmflg* are currently set in the shm_perm.mode field of the shared memory descriptor for the segment. If any mode bit set in *shmflg* is not set in shm_perm.mode, an error is returned. This is not an access check, because the process calling shmget requires no access to anything and the *shmflg* mode bits passed can all be zero. Rather, it guarantees that the shared memory segment is accessible for the access modes that may be desired by the caller.

ACCESS CONTROL

No access permission is required to do a shmget operation (except for the consistency check made on *shmflg*).

RETURN VALUE

shmid A non-negative integer, namely a shared memory identifier indicating the shmget operation was successful.

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EINVAL A shared memory identifier was to be created but *size* is less than the system-imposed minimum or greater than the system-imposed maximum; or a shared memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to zero.

EACCES A shared memory identifier exists for *key* but one of the low-order 9 bits set in *shmflg* is not set in the shm_perm.mode field of the shared memory identifier's corresponding shared memory descriptor.

ENOENT A shared memory identifier does not exist for *key* and the "create if not already existing" option was not selected, that is, the IPC_CREAT option was not specified in *shmflg*.

ENOSPC A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory identifiers system wide would be exceeded. Another shared memory segment cannot be created until one is destroyed.

ENOMEM A shared memory identifier and associated shared memory segment are to be created but there is not enough internal system memory available to fill the request. This is different from ENOSPC in that arbitrary operations that free internal system memory may allow the call to succeed at a later time.

EEXIST A shared memory identifier exists for *key* but the "create only if not already existing" option was selected, that is, the IPC_CREAT and IPC_EXCL options were on in *shmflg*.

SEE ALSO

intro(2), ipcrm(1), ipcs(1), shmctl(2), stdipc(3C).

NAME

`shmsys` – perform a shared memory operation

SYNOPSIS

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

int  shmsys (P1, P2, P3, P4)
int  P1;
int  P2;
int  P3;
int  P4;
```

where:

P1 An integer indicating the type of operation to be performed with shared memory (0 = SHMCTL, 1 = SHMGET, 2 = SHMAT, 3 = SHMDT)

P2 In case of SHMCTL or SHMAT, *P2* is a shared memory id. In case of SHMGET, *P2* is a shared memory key. In case of SHMDT, *P2* is a shared memory segment address.

P3 In case of SHMCTL, *P3* is a control command. In case of SHMGET, *P3* is the shared memory segment size. In case of SHMAT, *P3* is the shared memory segment address.

P4 In case of SHMCTL, *P4* is a pointer to a buffer, which contains all the information about the shared memory segment. In case of SHMGET and SHMAT, *P4* is a flag.

DESCRIPTION

The `shmsys` system call performs a shared memory operation (SHMCTL, SHMGET, SHMAT, SHMDT) indicated by the value of *P1*.

ACCESS CONTROL

See the descriptions of the exception condition EACCES in the man pages for the `shmget`, `shmctl`, `shmat`, and `shmdt` system calls.

RETURN VALUE

shmid If SHMGET was successful.

shmaddr If SHMAT was successful.

0 If SHMCTL or SHMDT was successful.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

The error codes returned depend on the type of shared memory operations performed and are described in `shmget`, `shmctl`, `shmat`, `shmdt`.

EINVAL *P1* argument is not in the range of 0 through 3.

SEE ALSO

`intro`(2), `shmget`(2), `shmctl`(2), `shmat`(2), `shmdt`(2).

NAME

`shutdown` – shut down part of a full-duplex connection

SYNOPSIS

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

where:

s File descriptor of socket to shut down

how Flag (0, 1, or 2) for what to shut down

DESCRIPTION

The `shutdown` call shuts down all or part of a full-duplex connection on the socket associated with *s*. If *how* is 0, then further receives will be disallowed. If *how* is 1, then further sends will be disallowed. If *how* is 2, then further sends and receives will be disallowed.

ACCESS CONTROL

None.

RETURN VALUE

0 Completed successfully.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF
The argument *s* is not an active valid descriptor.

ENOTSOCK
s is a file, not a socket.

ENOTCONN
The specified socket is not connected.

EINVAL
The *how* parameter is out of range.

SEE ALSO

`connect(2)`, `socket(2)`.

NAME

sigaction – examine and change signal action

SYNOPSIS

```
#include <signal.h>

int    sigaction (sig, act, oact)
int    sig;
const struct sigaction *act;
struct sigaction *oact;
```

where:

sig A signal number.

act NULL, or a new action to be installed for *sig*.

oact NULL, or the current action associated with *sig*. If *act* is not NULL and the call is successful, *oact* will be replaced by *act*.

DESCRIPTION

The sigaction() function allows the calling process to examine or specify (or both) the action to be associated with a specific signal. The argument *sig* specifies the signal; acceptable values are defined in <signal.h>.

The structure *sigaction*, used to describe an action to be taken, is defined in the header <signal.h> and includes the following members:

Member Type	Member Name	Description
void (*)()	sa_handler	SIG_DFL, SIG_IGN, or pointer to a function.
sigset_t	sa_mask	Additional set of signals to be blocked during execution of signal-catching function.
int	sa_flags	Special flags to affect behavior of signal.

If the argument *act* is not NULL, it points to a structure specifying the action to be associated with the specified signal. If the argument *oact* is not NULL the action previously associated with the signal is stored in the location pointed to by the argument *oact*. If the argument *act* is NULL signal handling is unchanged by this function call; thus, the call can be used to enquire about the current handling of a given signal.

The *sa_handler* field of the *sigaction* structure identifies the action to be associated with the specified signal. It may have any of the values specified above.

If the *sa_handler* field specifies a signal-catching function, the *sa_mask* field identifies a set of signals that shall be added to the process's set of blocked signals before the signal-catching function is invoked. In addition, the signal that caused the handler to be invoked will be added to the set of blocked signals unless the SA_NODEFER flag has been specified (see the *sa_flags* description below). The SIGKILL and SIGSTOP signals shall not be added to the signal mask using this mechanism; this restriction shall be enforced by the system without causing an error to be indicated.

The *sa_flags* field can be used to modify the delivery of the specified signal.

The following flags, defined in the header <signal.h>, can be set in *sa_flags*:

Symbolic Constant	Description
SA_ONSTACK	If set and the signal is caught, and an alternate signal stack has been declared, the signal is delivered to the calling process using the alternate stack. Otherwise, the signal is delivered on the same stack as the main program.
SA_RESETHAND	If set and the signal is caught, the action of the signal is reset to SIG_DFL. (Note: SIGILL, SIGTRAP, and SIGPWR cannot be automatically reset when delivered; this restriction shall be enforced by the system without causing an error to be indicated.)
SA_NODEFER	If set and the signal is caught, *sig* will not be automatically blocked when while the handler is active.
SA_RESTART	If set and the signal is caught, a restartable system call that is interrupted by the execution of the signal's handler will be transparently restarted when the handler finishes. If this flag is not set, a system call that is interrupted will return EINTR.
SA_SIGINFO	If this flag is not set and the signal is caught, *sig* is passed as the only argument to the signal handling function. If this flag is set and the signal is caught, two additional arguments will be passed to the signal handling function. If the second argument is not equal to NULL, it will point to an object of type siginfo_t, which will explain the reason the signal was generated (see `siginfo.h`). The third argument will point to an object of type ucontext_t, which will describe the receiving process' context at the time it received the signal (see `ucontext.h`).
SA_NOCLDWAIT	If set and *sig* equals SIGCHLD, the system will clean up after the calling processes dead children. If the calling process subsequently calls `wait()`, it will block until all of its processes terminate and then return a value of −1 with `errno` set to ECHLD.
SA_NOCLDSTOP	If set and *sig* equals SIGCHLD, *sig* will not be sent to the calling process when its child processes stop.

When a signal is caught by a signal-catching function installed by the `sigaction()` function, a new signal mask is calculated and installed for the duration of the signal-catching function (or until a call to either the `sigprocmask()` or `sigsuspend()` function is made). This mask is formed by taking the union of the current signal mask and the value of the *sa_mask* for the signal being delivered, and then including the signal being delivered (unless the SA_NODEFER flag is set, as described above). If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested (by another call to the `sigaction()` function), or until one of the exec() functions is called. This behavior may be modified by using the SA_RESTART flag as described above.

If the previous action for *sig* had been established by the signal() function, defined in the C Standard, the values of the fields returned in the structure pointed to by *oact* are unspecified, and in particular *oact->sv_handler* is not necessarily the same value passed to the signal() function. However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to the sigaction() function via the *act* argument, handling of the signal shall be as if the original call to the signal() function were repeated.

If the sigaction() function fails, no new signal handler is installed.

RETURN VALUE

0 Successful completion.

−1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

If any of the following conditions occur, the sigaction() function shall return −1 and set errno to the corresponding value:

EINVAL The value of the *sig* argument is an invalid or unsupported signal number, or an attempt was made to catch a signal that cannot be caught or to ignore a signal that cannot be ignored. See <signal.h>.

EFAULT *act* or *oact* points to an invalid location in the user's address space.

SEE ALSO

kill(2), sigprocmask(2), sigsuspend(2), sigsetops(3C), <signal.h>.

COPYRIGHTS

NAME

sigaltstack – set or get signal alternate stack context

SYNOPSIS

```
#include &lt;signal.h&gt;

int sigaltstack(const stack_t *ss, stack_t *oss);
```

where:

ss A structure specifying the new alternate signal stack

oss A structure specifying the old alternate signal stack

DESCRIPTION

sigaltstack allows users to define an alternate stack area on which signals are to be processed. If *ss* is non-zero, it specifies a pointer to, and the size of a stack area on which to deliver signals, and tells the system if the process is currently executing on that stack. When a signal's action indicates its handler should execute on the alternate signal stack [specified with a call to sigaction(2) or sigvec(2)], the system checks to see if the process is currently executing on that stack. If the process is not currently executing on the signal stack, the system arranges a switch to the alternate signal stack for the duration of the signal handler's execution.

The structure sigaltstack includes the following members.

```
int     *ss_sp
long    ss_size
int     ss_flags
```

If *ss* is not NULL, it points to a structure specifying the alternate signal stack that will take effect upon return from sigaltstack. The ss_sp and ss_size fields specify the new base and size of the stack, which is automatically adjusted for direction of growth and alignment. The ss_flags field specifies the new stack state and may be set to the following:

SS_DISABLE The stack is to be disabled and ss_sp and ss_size are ignored. If SS_DISABLE is not set, the stack will be enabled.

If *oss* is not NULL, it points to a structure specifying the alternate signal stack that was in effect prior to the call to sigaltstack. The ss_sp and ss_size fields specify the base and size of that stack. The ss_flags field specifies the stack's state, and may contain the following values:

SS_ONSTACK The process is currently executing on the alternate signal stack. Attempts to modify the alternate signal stack while the process is executing on it will fail.

SS_DISABLE The alternate signal stack is currently disabled.

ACCESS CONTROL

No access checking is performed.

RETURN VALUE

On success, sigaltstack returns zero. On failure, it returns –1 and sets errno to indicate the error.

DIAGNOSTICS

EINVAL *ss* is non-null and its *ss_flags* field has one or more invalid flags.

EPERM An attempt was made to modify an active stack.

ENOMEM The size of the alternate stack area is less than MINSIGSTKSZ.

EFAULT Either *ss* or *oss* points to memory which is not a valid part of the proces's address space.

SEE ALSO

getcontext(2), sigaction(2), sigvec(2), sigsetjmp(3C), ucontext(5).

NOTES

The value SIGSTKSZ is defined to be the number of bytes that would be used to cover the usual case when allocating an alternate stack area. The value MINSIGSTKSZ is defined to be the minimum stack size for a signal handler. In computing an alternate stack size, a program should add that amount to its stack requirements to allow for the operating system overhead.

The following code fragment is typically used to allocate an alternate stack.

```
if ((sigstk.ss_sp = (char *)malloc(SIGSTKSZ)) == NULL)
     /* error return */;

sigstk.ss_size = SIGSTKSZ;
sigstk.ss_flags = 0;
if (sigaltstack(&sigstk, (stack_t *)0) < 0)
     perror("sigaltstack");
```

NAME

sigblock – add to set of blocked signals

SYNOPSIS

```
#include <signal.h>

long  sigblock (signal_mask)
long  signal_mask;
```

where:

signal_mask Set of additional signals to block

DESCRIPTION

`Sigblock` adds the set of signals specified in *signal_mask* to the set of signals currently being blocked from presentation. Signal *s* is represented by the value `sigmask(s)` in *signal_mask*.

It is not possible to block SIGKILL, SIGSTOP, or SIGCONT. It may or may not be possible to block signals that are not defined by the system. An attempt to block these signals will not produce an error.

ACCESS CONTROL

None.

RETURN VALUE

old_signal_mask The previous set of signals being blocked from presentation.

DIAGNOSTICS

None.

SEE ALSO

`kill`(2), `sigvec`(2), `sigsetmask`(2).

NAME

`sigfillset` – fill in the set of implementation-defined signals

SYNOPSIS

```
int        sigfillset(signal_mask)
sigset_t *signal_mask;
```

where:

signal_mask A pointer to a signal mask

DESCRIPTION

The `sigfillset` call sets the signal mask pointed to by *signal_mask* to contain all signals defined in this implementation.

RETURN VALUE

0 The operation was successful.

–1 The operation was not successful.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EFAULT The argument *signal_mask* specifies an invalid area of the calling process's address space or an area which does not have write access.

SEE ALSO

`kill(2)`, `signal(2)`.

NAME

`sighold` – add a signal to the calling process's set of blocked signals

SYNOPSIS

```
#include <signal.h>

int  sighold (signal_number)
int  signal_number;
```

where:

signal_number The signal to be blocked

DESCRIPTION

`Sighold` adds the specified signal to the calling process's set of signals blocked from presentation. If the specified signal is already blocked, no error is reported, but this call has no effect as block operations do not nest.

It is not possible to block SIGKILL, SIGSTOP, or SIGCONT. It may or may not be possible to block signals that are not defined by the system. An attempt to block these signals will produce the error EINVAL.

Note that this system call performs exactly the same basic operation as the system call `sigset` with the function parameter set to SIG_HOLD and as the system call `sigblock` with a mask specifying a single signal. These three system calls differ in their return values and in reporting attempts to block a signal that cannot be blocked.

ACCESS CONTROL

None.

RETURN VALUE

0 The operation succeeded.

–1 The operation failed.

DIAGNOSTICS

`Errno` may be set to the following error code:

EINVAL *Signal_number* is an illegal signal number or one which may not be blocked.

SEE ALSO

`sighold(2), sigignore(2), sigpause(2), sigrelse(2), sigset(2).`

NAME

`sigignore` – set the signal action of a signal to 'ignore'

SYNOPSIS

```
#include <signal.h>

int  sigignore (signal_number)
int  signal_number;
```

where:

signal_number The signal whose action is to be changed to ignore

DESCRIPTION

`Sigignore` sets the signal action associated with the specified signal to 'ignore' and the signal is removed from the set of signals blocked from presentation. (Any pended signals are also effectively discarded because as soon as they are unblocked, they are ignored.)

It is not possible to ignore SIGKILL, SIGSTOP, or SIGCONT (see `sys/signal.h`). An attempt to ignore these signals will produce the error EINVAL.

This system call performs exactly the same basic operation as the `signal`, `sigset`, and `sigvec` system calls with the function set to SIG_IGN. Note, however, that `sigvec` does NOT remove the signal from the set of blocked signals.

ACCESS CONTROL

None.

RETURN VALUE

0 The operation succeeded.

–1 The operation failed.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EINVAL *Signal_number* is an illegal signal number or a signal that may not be ignored.

SEE ALSO

`sighold(2)`, `sigignore(2)`, `sigpause(2)`, `sigrelse(2)`, `sigset(2)`.

NAME

signal – specify what to do upon presentation of a signal

SYNOPSIS

```
#include <signal.h>

void  (* signal (signal_number, action))()
int   signal_number;
void  (*action)();
```

where:

signal_number Any of the valid signals except SIGKILL (see <sys/signal.h>, which is included into <signal.h>, for a complete list)

action Handler for the signal: SIG_DFL, SIG_IGN, or a function address

DESCRIPTION

This manual page describes the default signal behavior. If you define the _BSD_SIGNAL_FLAVOR macro or if you define only the _BSD_SOURCE macro when you compile your C application, however, you will get the behavior described in berk_signal(3C) (also found as signal(3C)). For more information about the _BSD_SIGNAL_FLAVOR and _BSD_SOURCE macros and the capabilities they provide, see *Porting Applications to the DG/UX™ System*.

Signal allows the calling process to choose one of three ways to handle the presentation of a specific signal. *Signal_number* specifies the signal, and *action* specifies the choice. The actions prescribed by *action* are as follows.

SIG_DFL Terminate the process.

The process's signal action vector entry for *signal_number* is set to 'default'. If the signal *signal_number* was pended and *signal_number* is not SIGKILL, the pended signal is lost. The set of blocked signals remains unchanged.

When the signal *signal_number* is presented to the process, it will cause the process either to terminate, stop, ignore the signal, or terminate with a core dump depending on the signal's type (see <sys/signal.h>).

If a core dump is indicated, the receiving process must have adequate permission to do so.

SIG_IGN Ignore signal.

The process's signal action vector entry for *signal_number* is set to 'ignore'. The set of blocked signals remains unchanged.

When the signal *signal_number* is presented to the process, it will be discarded.

SIGKILL cannot be ignored.

address Catch signal.

The process's signal action vector entry for *signal_number* is set to 'catch'. If the signal *signal_number* was pended and *signal_number* is not SIGKILL, the pended signal is lost. The set of blocked signals remains unchanged.

signal 10 SIGBUS

When the signal *signal_number* is sent to the process, it will cause the signal handler specified by *action* to be invoked.

The following attributes are set for the signal action vector entry for *signal_number*:

- The signal mask addend is cleared. Thus, no additional signals will be blocked when the signal handler is invoked.
- The signal stack choice specifies the current execution stack. Thus, no stack change is made.
- If *signal_number* is not `SIGILL`, `SIGTRAP`, or `SIGPWR`, the system first sets the signal action to `SIG_DFL` before executing the signal handler. For signals whose new signal action is set to `SIG_DFL`, the occurrence of multiple signals may cause some signals to be lost or may cause the process to terminate.
- System calls interrupted by signal *signal_number* will not be restarted.

  The value of the signal *action* is not verified or access checked at the time of the call. If it is invalid, results are undefined when the signal is caught.

  SIGKILL cannot be caught.

After a fork, the child process inherits all software signal structures, except that the pending signal vector is cleared.

`Exec` modifies the software signal structures in the following manner:

1) The signal action for signals set to 'catch' is changed to 'default'.

2) The signal stack context is discarded.

3) All other software signal structures are unchanged.

Setting the signal SIGCLD to SIG_IGN affects `exit` and `wait` in the following manner:

1) The calling process's child processes will be cleaned up by the parent when the parent issues its next system call (checks signals).

2) If the calling process later performs a `wait` operation, `wait` will suspend the calling process until all child processes have terminated and will return with the error condition ECHILD.

`Signal` will fail, and the signal handler will be unchanged if an error occurs.

ACCESS CONTROL

No access is required to install a signal handler.

The receiving process is granted permission to produce a core dump file provided:

- the effective-user-id and the real-user-id of the receiving process are equal, and
- the receiving process has adequate file system permission to create or rewrite the core dump file.

RETURN VALUE

old_action Completed successfully. The previous signal handler for *signal_number* is returned.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to the following error code:

EINVAL *Signal_number* is an illegal signal number, including SIGKILL.

SEE ALSO

`kill`(2), `pause`(2), `ptrace`(2), `wait`(2), `berk_signal`(3C).

STANDARDS

When using `m88kbcs` as the Software Development Environment target, the `signal` function will be emulated using BCS system calls. Since this is an emulation requiring several BCS system calls, a slight performance degradation may be noticed in comparison to using `signal` in `/lib/libc.a`.

NAME

sigpause – clear a blocked signal and suspend the process until a signal is caught

SYNOPSIS

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

int  sigpause (signal_number)
int  signal_number;
```

where:

signal_number The signal whose blocked state is to be cleared

DESCRIPTION

Sigpause removes the specified signal from the set of signals blocked from presentation and then suspends the caller until a signal is caught.

This function is exactly equivalent to the system call sigrelse followed by pause.

ACCESS CONTROL

None.

RETURN VALUE

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EINVAL *signal_number* is an illegal signal number or a signal that cannot be unblocked.

EINTR A signal interrupted the sigpause operation

SEE ALSO

berk_sigpause(2), kill(2), pause(2), sighold(2), sigignore(2), signal(2), sigrelse(2), sigset(2).

STANDARDS

When using m88kbcs as the Software Development Environment target, the sigpause function will be emulated using BCS system calls. Since this is an emulation requiring several BCS system calls, a slight performance degradation may be noticed in comparison to using sigpause in /lib/libc.a.

NAME

`sigpending` – examine pending signals

SYNOPSIS

```
#include <signal.h>

int sigpending (set)
sigset_t *set;
```

where:

set A structure to which the list of signals are to be written

DESCRIPTION

The `sigpending()` function shall store the set of signals that are blocked from delivery and pending for the calling process, in the space pointed to by the argument *set*.

RETURN VALUE

0 Successful completion.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

This standard does not specify any error conditions that are required to be detected for the `sigpending()` function. Some errors may be detected under implementation-defined conditions.

SEE ALSO

`sigprocmask`(2), `sigsetops`(3C), `<signal.h>`.

COPYRIGHTS

STANDARDS

The EFAULT error will be generated if the argument *set* specifies an invalid error of the calling process's address space, or an address area which does not have write access.

NAME

`sigprocmask` – examine and change blocked signals

SYNOPSIS

```
#include <signal.h>

int sigprocmask (how, set, oset)
int how;
const sigset_t *set;
sigset_t       *oset;
```

where:

how The manner in which the current set of blocked signals is changed.

set NULL, or the signal set used to change the current set of blocked signals.

oset NULL, or the current set of blocked signals.

DESCRIPTION

The `sigprocmask` function is used to examine or change (or both) the calling process's signal mask. If the value of the argument *set* is not NULL, it points to a set of signals to be used to change the currently blocked set.

The value of the argument *how* indicates the manner in which the set is changed, and shall consist of one of the following values, as defined in the header `<signal.h>`.

Name	Description
SIG_BLOCK	The resulting set shall be the union of the current set and the signal set pointed to by the argument *set*.
SIG_UNBLOCK	The resulting set shall be the intersection of the current set and the complement of the signal set pointed to by the argument *set*.
SIG_SETMASK	The resulting set shall be the signal set pointed to by the argument *set*.

If the argument *oset* is not NULL, the previous mask is stored in the space pointed to by *oset*. If the value of the argument *set* is NULL, the value of the argument *how* is not significant and the process's signal mask is unchanged by this function call; thus, the call can be used to enquire about currently blocked signals.

If there are any pending unblocked signals after the call to the `sigprocmask` function, at least one of those signals shall be delivered before the `sigprocmask` function returns.

It is not possible to block the SIGKILL and SIGSTOP signals; this shall be enforced by the system without causing an error to be indicated.

If any of the SIGFPE, SIGILL, or SIGSEGV signals are generated while they are blocked, the result is undefined, unless the signal was generated by a call to the `kill` function or the `raise` function defined by the C Standard.

If the `sigprocmask` function fails, the process's signal mask is not changed by this function call.

RETURN VALUE

0 Successful completion.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

If any of the following conditions occur, the `sigprocmask` function shall return −1 and set `errno` to the corresponding value:

EINVAL The value of the *how* argument is not equal to one of the defined values.

SEE ALSO

`sigaction`(2), `sigpending`(2), `sigsetops`(3C), `sigsuspend`(2), `<signal.h>`.

COPYRIGHTS

NAME

sigrelse – remove a signal from the calling process's set of blocked signals

SYNOPSIS

```
#include <signal.h>

int  sigrelse (signal_number)
int  signal_number;
```

where:

signal_number The signal to be removed from the set of blocked signals

DESCRIPTION

Sigrelse removes the specified signals from the calling process's set of signals blocked from presentation. If the specified signal is not currently blocked, no error is reported, but this call has no effect as block/unblock operations do not nest.

It is not possible to unblock SIGKILL, SIGSTOP, or SIGCONT. It may or may not be possible to unblock signals that are not defined by the system. An attempt to unblock these signals will produce the error EINVAL.

ACCESS CONTROL

None.

RETURN VALUE

0 The operation succeeded.

–1 The operation failed.

DIAGNOSTICS

Errno may be set to the following error code:

EINVAL *Signal_number* is an illegal signal number or one which may not be unblocked.

SEE ALSO

sighold(2), sigignore(2), sigpause(2), sigrelse(2), sigset(2).

NAME

sigret – restore the process state to that contained in a signal frame

SYNOPSIS

```
#include <signal.h>

void  sigret ()
```

DESCRIPTION

The `sigret` call restores the process state to that contained in a signal frame pointed to by r31. These values (with the exception of the four modifiable fields, SXIP, SNIP, SFIP, and r31) must be identical to the values in a signal frame that was pushed onto the user's stack when the process's signal handler was invoked (that is, this is a "return from signal handler").

ACCESS CONTROL

None.

RETURN VALUE

This function never returns. Execution resumes at the point specified by the `sigcontext` structure. If an error occurs, this function terminates the process with an exit status which indicates that the process was killed by a SIGSEGV signal.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EFAULT	The `sigcontext` structure pointed to by r31 could not be accessed.
EINVAL	The `sigcontext` structure pointed to by r31 contains invalid information.

SEE ALSO

`kill(2)`, `ptrace(2)`, `sigblock(2)`, `sigpause(2)`, `sigsetmask(2)`, `sigstack(2)`, `sigvec(2)`.

NAME

`sigsend`, `sigsendset` – send a signal to a process or a group of processes

SYNOPSIS

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

int sigsend(idtype_t idtype, id_t id, int sig);

int sigsendset(procset_t *psp, int sig);
```

where:

idtype	`P_PID`, `P_PGID`, `P_SID`, `P_UID`, `P_GID`, `P_CID`, or `P_ALL`,
id	An identification number or `P_MYID`
psp	A structure of type `procset_t` defined in `procset.h`
sig	A number or name specifying a signal

DESCRIPTION

`Sigsend` sends a signal to the process or group of processes specified by *id* and *idtype*. The signal to be sent is specified by *sig* and is either zero or one of the values listed in `signal`(5). If *sig* is zero (the null signal), error checking is performed but no signal is actually sent. This value can be used to check the validity of *id* and *idtype*.

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, or *sig* is `SIGCONT` and the sending process has the same session ID as the receiving process.

If *idtype* is `P_PID`, *sig* is sent to the process with process ID *id*.

If *idtype* is `P_PGID`, *sig* is sent to any process with process group ID *id*.

If *idtype* is `P_SID`, *sig* is sent to any process with session ID *id*.

If *idtype* is `P_UID`, *sig* is sent to any process with effective user ID *id*.

If *idtype* is `P_GID`, *sig* is sent to any process with effective group ID *id*.

If *idtype* is `P_CID`, *sig* is sent to any process with scheduler class ID *id*.

If *idtype* is `P_ALL`, *sig* is sent to all processes and *id* is ignored.

If *id* is `P_MYID`, the value of *id* is taken from the calling process.

The process with a process ID of 0 is always excluded. The process with a process ID of 1 is excluded unless *idtype* is equal to `P_PID`.

`sigsendset` provides an alternate interface for sending signals to sets of processes. This function sends signals to the set of processes specified by *psp*. *psp* is a pointer to a structure of type `procset_t`, defined in `<sys/procset.h>`, which includes the following members:

```
idop_t      p_op;
idtype_t    p_lidtype;
id_t        p_lid;
idtype_t    p_ridtype;
id_t        p_rid;
```

`p_lidtype` and `p_lid` specify the ID type and ID of one ("left") set of processes; `p_ridtype` and `p_rid` specify the ID type and ID of a second ("right") set of processes. ID types and IDs are specified just as for the *idtype and id* arguments to sigsend. p_op specifies the operation to be performed on the two sets of processes to get the set of processes the system call is to apply to. The valid values for p_op and the processes they specify are:

POP_DIFF	set difference: processes in left set and not in right set
POP_AND	set intersection: processes in both left and right sets
POP_OR	set union: processes in either left or right set or both
POP_XOR	set exclusive-or: processes in left or right set but not in both

ACCESS CONTROL

If user ID of the sending process is not superuser, then its real or effective user ID must match the real or effective user ID of the receiving process(es), unless it is sending SIGCONT to a process in that shares its session.

RETURN VALUE

On success, sigsend sigsendset return zero. On failure, it returns −1 and sets errno to indicate the error.

Errno may be set to the following error code:

EINVAL	*sig* is not a valid signal number.
EINVAL	*idtype* is not a valid idtype field.
EINVAL	*sig* is SIGKILL, *idtype* is P_PID and *id* is 1 (proc1).
ESRCH	No process can be found corresponding to that specified by *id* and *idtype*.
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, and the calling process is not sending SIGCONT to a process that shares the same session.

In addition, sigsendset may set errno to

EFAULT	psp points outside the process's allocated address space.

SEE ALSO

getpid(2), getpgrp(2), kill(1), kill(2), setpid(2), signal(2), signal(5).

NAME

`sigset` – specify what to do upon presentation of a signal

SYNOPSIS

```
#include <signal.h>

void  (* sigset (signal_number, action))()
int   signal_number;
void  (*action)();
```

where:

signal_number Any of the valid signals except SIGKILL (see `signal.h` for a complete list)

action Handler for the signal: SIG_DFL, SIG_IGN, SIG_HOLD, or a function address

DESCRIPTION

`Sigset` lets the calling process choose one of four ways to handle the presentation of a specific signal. *signal_number* specifies the signal and *action* specifies the choice. The action choices are as follows:

SIG_DFL Process termination.

The process's signal action vector entry for *signal_number* is set to 'default' and the blocked signal vector entry for *signal_number* is cleared.

When the signal *signal_number* is sent to the process, it will not be pended and will cause the process to either terminate, stop, ignore the signal, or terminate with a core dump depending on the signal's type (see `signal.h`).

If a core dump is indicated, the receiving process must have adequate permission to do so.

SIG_IGN Ignore signal.

The process's signal action vector entry for *signal_number* is set to 'ignore' and the blocked signal vector entry for *signal_number* is cleared.

When the signal *signal_number* is sent to the process, it will not be pended and will be discarded.

SIGKILL, SIGCONT and SIGSTOP cannot be ignored.

SIG_HOLD Hold signal.

The process's signal action vector entry for *signal_number* is not modified, but the block signal vector entry for *signal_number* is set. This option is equivalent to the system call `sighold`.

address Catch signal.

The process's signal action vector entry for *signal_number* is set to 'catch' and the blocked signal vector entry for *signal_number* is cleared. If the signal *signal_number* was pended, the signal is presented to the process.

When the signal *signal_number* is sent to the process, it will not be pended and will cause signal handler specified by *action* to be invoked.

The following attributes are set for the signal action vector entry for *signal_number*:

- The signal mask addend is set to the specified signal. Thus, the specified signal will be added to the set of blocked signals when the signal handler is invoked.
- The signal stack choice specifies the current execution stack. Thus, no stack change is made.
- System calls interrupted by signal *signal_number* will not be restarted.

SIGKILL cannot be caught.

After a fork, the child process inherits all software signal structures, except that the pending signal vector is cleared.

`Exec` modifies the software signal structures in the following manner: 1) The signal action for signals set to 'catch' is changed to 'default'. 2) The signal stack context is discarded. 3) All other software signal structures are unchanged.

Setting the signal SIGCLD to SIG_IGN affects exit and wait in the following manner: 1) The calling process's child processes will be cleaned-up by the system when they terminate. 2) If the calling process later performs a wait operation, wait will suspend the calling process until all child processes have terminated and will return with the error condition ECHILD.

Signal will fail and the signal handler will be unchanged if an error occurs.

ACCESS CONTROL

No access is required to install a signal handler.

The receiving process is granted permission to produce a core dump file provided

- the effective-user-id and the real-user-id of the receiving process are equal, and
- the receiving process has adequate file system permission to create or rewrite the core dump file.

RETURN VALUE

old_action	Completed successfully. The previous signal handler for *signal_number* is returned.
SIG_ERR	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to the following error code:

EINVAL	*Signal_number* is an illegal signal number, including SIGKILL.

SEE ALSO

`kill`(2), `pause`(2), `ptrace`(2), `wait`(2).

STANDARDS

When using `m88kbcs` as the Software Development Environment target, the `sigset` function will be emulated using BCS system calls. Since this is an emulation requiring several BCS system calls, a slight performance degradation may be noticed in comparison to using `sigset` in `/lib/libc.a`.

NAME

sigsetmask – specify set of blocked signals

SYNOPSIS

```
#include <signal.h>

int  sigsetmask (signal_mask)
int  signal_mask;
```

where:

signal_mask Set of signals to be blocked

DESCRIPTION

`Sigsetmask` assigns the set of signals specified in *signal_mask* to the set of signals blocked from presentation. Signal *s* is represented by the value `sigmask(`*s*`)` in *signal_mask*.

It is not possible to block SIGKILL, SIGSTOP, or SIGCONT. It may or may not be possible to block signals that are not defined by the system. An attempt to block these signals will not produce an error.

ACCESS CONTROL

None.

RETURN VALUE

old_signal_mask The previous set of signals being blocked from presentation.

DIAGNOSTICS

None.

SEE ALSO

kill(2), sigblock(2), sigpause(2), sigvec(2).

NAME

sigstack – set and/or get signal stack context

SYNOPSIS

```
#include <signal.h>

int    sigstack (new_signal_stack, old_signal_stack)
struct sigstack * new_signal_stack;
struct sigstack * old_signal_stack;
```

where:

new_signal_stack	NULL or address of new signal stack context specifier
old_signal_stack	NULL or address of old signal stack context specifier

DESCRIPTION

Sigstack is used to install a new signal stack context and retrieve the previous signal stack context. A new signal stack context is optionally installed using the *new_signal_stack* parameter. If *new_signal_stack* is NULL, the signal stack context remains unchanged. Otherwise, *new_signal_stack* is installed. The previous signal stack context may be obtained by the *old_signal_stack* parameter. If *old_signal_stack* is NULL, the previous signal stack context is not returned. Otherwise, the previous signal stack context information is stored in the location pointed to by *old_signal_stack*.

A signal stack is an alternate execution stack on which signals are processed. The signal stack context consists of two components: the address of the base of the signal stack (ss_sp) and an indication as to whether the process is currently executing on the signal stack (ss_onstack).

In DG/UX, the user's stack grows from high to low addresses. Therefore, the stack pointer, *ss_sp*, must be the upper bound of the memory allocated for the alternate signal stack. The caller must make this adjustment; it will not be made by the system.

When a signal's action is 'catch' and its signal stack choice specifies the signal stack, the system checks to see if the process is currently executing on the signal stack. If the process is not currently executing on the signal stack, the system arranges a switch to the signal stack for the duration of the signal handler.

Signal stacks do not increase automatically, as is done for the normal stack. If the stack overflows unpredictable results may occur.

Sigstack will fail and the signal stack context will be unchanged if an error occurs.

ACCESS CONTROL

None.

RETURN VALUE

0	Completed successfully.
–1	An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to the following error code:

EFAULT	Either *new_signal_stack* or *old_signal_stack* points to memory which is not a valid part of the process address space. The validity of the signal stack is not checked.

SEE ALSO

sigvec(2), setjmp(3C).

NAME

sigsuspend – wait for a signal

SYNOPSIS

```
#include <signal.h>

int sigsuspend (sigmask)
sigset_t *sigmask;
```

where:

sigmask A structure containing a set signals constituting a signal mask

DESCRIPTION

The sigsuspend() function replaces the process's signal mask with the set of signals pointed to by the argument *sigmask* and then suspends the process until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

If the action is to terminate the process, the sigsuspend() function shall not return. If the action is to execute a signal-catching function, the sigsuspend() shall return after the signal-catching function returns, with the signal mask restored to the set that existed prior to the sigsuspend() call.

It is not possible to block those signals that cannot be ignored, as documented in <signal.h> this shall be enforced by the system without causing an error to be indicated.

RETURN VALUE

Since the sigsuspend() function suspends process execution indefinitely, there is no successful completion return value. A value of –1 is returned and errno is set to indicate the error.

DIAGNOSTICS

If any of the following conditions occur, the sigsuspend() function shall return –1 and set errno to the corresponding value:

EINTR A signal is caught by the calling process and control is returned from the signal-catching function.

SEE ALSO

pause(2), sigaction(2), sigpending(2), sigprocmask(2), sigsetops(3C), <signal.h>.

COPYRIGHTS

NAME

sigvec – specify what to do upon presentation of a signal

SYNOPSIS

```
#include <signal.h>

int sigvec (signal_number, new_signal_vector, old_signal_vector)
int signal_number;
struct sigvec * new_signal_vector;
struct sigvec * old_signal_vector;
```

where:

signal_number	Any of the valid signals except SIGKILL or SIGSTOP (see `signal.h` for a complete list)
new_signal_vector	NULL or address of new handler specifier
old_signal_vector	NULL or address of old handler specifier

DESCRIPTION

`Sigvec` is used to install a new handler and retrieve the previous handler for signal *signal_number*. A handler for the signal is optionally installed using the *new_signal_vector* parameter. If *new_signal_vector* is NULL, the handler remains unchanged. Otherwise, *new_signal_vector* is installed. The previous handler for the signal may be obtained by the *old_signal_vector* parameter. If *old_signal_vector* is NULL, the previous handler is not returned. Otherwise, the previous handler information is stored in the location pointed to by *old_signal_vector*.

A signal handler has three components: a set of flags (`sv_flags`), a signal mask (`sv_mask`), and an action (`sv_handler`).

Each signal handler may choose to execute on either the current stack of the calling process or on a special signal stack. The process must have previously defined the signal stack using `sigstack`. The handler's stack choice is indicated by a flag in `sv_flags`. Setting the flag SV_ONSTACK chooses the signal stack of the calling process; otherwise the current stack is used. The stack address is chosen when the signal is presented. Thus, subsequent sigstack operations may redirect the handler's signal stack.

The handler's signal mask is an additional set of signals that are to be blocked from presentation while the signal is being handled. The set of signals that are blocked while the signal is being handled is the union of the handler's signal mask, the signal that occurred, and the process's current set of blocked signals.

Signal *s* is represented by the value `sigmask(`*s*`) in sv_mask`.

The handler's action chooses one of three ways to handle the receipt of a signal. *new_signal_vector*`.sv_handler` may be assigned one of the values: SIG_DFL, SIG_IGN, or a function address. The actions prescribed by these values are as follows:

SIG_DFL Default signal action.

The process's signal action vector entry for *signal_number* is set to 'default'.

When the signal *signal_number* is sent to the process, it may be pended depending on the state of the blocked signal vector. When the signal is presented to the process, it will cause the process to either terminate, stop, ignore the signal, or terminate with a core dump depending on the signal's type (see `signal.h`).

If a core dump is indicated, the receiving process must have adequate permission to do so.

SIG_IGN Ignore signal.

The process's signal action vector entry for *signal_number* is set to 'ignore'.

When the signal *signal_number* is sent to the process, it may be pended depending on the state of the blocked signal vector. When the signal is presented to the process, it will be discarded.

SIGKILL, SIGSTOP, and SIGCONT cannot be ignored.

address Catch signal.

The process's signal action vector entry for *signal_number* is set to 'catch'.

When the signal *signal_number* is sent to the process, it may be pended depending on the state of the blocked signal vector. When the signal is presented to the process, it will cause the signal handler specified by *action* to be invoked.

The following attributes are set for the signal action vector entry for *signal_number*:

- The signal mask addend is set to the union of *new_signal_vector*.`sv_mask` and *signal_number*. These signals are added to the blocked signal vector for the duration of the signal handler's invocation.
- The signal stack choice is set based on the flag SV_ONSTACK. This may cause a stack switch to take place for the duration of the signal handler's invocation.
- The new signal action is set to 'unchanged'. The occurrence of multiple signals will not cause the loss of signals or process termination.
- The restart system call choice is set based on the flag SV_INTERRUPT. If the flag is set, system calls interrupted by signal *signal_number* will be be terminated with `errno` set to EINTR rather than being restarted. If the flag is not set, restartable system call will be transparantly restarted when the signal handler returns.

SIGKILL and SIGSTOP cannot be caught.

After a fork, the child process inherits all software signal structures, except that the pending signal vector is cleared.

`Exec` modifies the software signal structures in the following manner: 1) The signal action for signals set to 'catch' is changed to 'default'. 2) The signal stack context is discarded. 3) All other software signal structures are unchanged.

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

Sigvec will fail and the signal handler will be unchanged if an error occurs.

ACCESS CONTROL

No access is required to install a signal handler.

The receiving process is granted permission to produce a core dump file provided

- the effective-user-id and the real-user-id of the receiving process are equal, and
- the receiving process has adequate file system permission to create or rewrite the core dump file.

RETURN VALUE

0 Completed successfully.

-1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EFAULT	Either *new_signal_vector* or *old_signal_vector* point to memory which is not a valid part of the process address space.
EINVAL	*Signal_number* is not a valid signal number.
EINVAL	An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.
EINVAL	An attempt is made to ignore SIGCONT.

SEE ALSO

kill(1), kill(2), ptrace(2), sigblock(2), sigpause(2), sigsetmask(2), sigstack(2), sigvec(2), setjmp(3C), tty(7).

NAME

socket – create an endpoint for communication

SYNOPSIS

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

int  socket (af, type, protocol)
int  af;
int  type;
int  protocol;
```

where:

af	Protocol family (domain)
type	Type of service desired
protocol	Optional protocol id (usually 0)

DESCRIPTION

Socket creates an endpoint for communication and returns a descriptor for the socket.

The *af* parameter specifies the domain in which the socket should be created. The domain determines the semantics of the service provided and affects what services are available. The domains available in the system are configuration dependent. Domains are identified by constants defined in sys/socket.h. All constants begin with PF_; examples are PF_UNIX and PF_INET. However, defining a domain in sys/socket.h does not imply the domain is configured in the current system.

The socket has the indicated type that specifies the semantics of communication. Socket types are defined in sys/socket.h as constants beginning with SOCK_; examples are SOCK_STREAM and SOCK_DGRAM.

A SOCK_STREAM type provides sequenced, reliable, two-way connection-based byte streams with an out-of-band data transmission mechanism. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a small, fixed maximum length). SOCK_RAW sockets provide access to internal network interfaces. The type SOCK_RAW is available only to the superuser.

The *protocol* optionally specifies a particular protocol to be assigned to the socket. If the user doesn't care which protocol in the domain supplies the service, a protocol of zero can be given and the domain will choose an appropriate protocol.

However, many protocols may exist and a user can specify a particular protocol by giving the protocol identifier in this manner. The protocol number to use depends on the communication domain in which communication is to take place; see the related documentation for a particular domain for more information about individual protocols.

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a connect call. Once connected, data may be transferred using read and write calls or some variant of the send and recv calls. When a session has been completed, a close may be performed. Out-of-band data may also be transmitted as described in send and received as described in recv.

The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken. Subsequent calls will return an error, –1. The specific error code in global variable `errno` will be `ETIMEDOUT`. The protocols optionally keep sockets warm by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g., five minutes). A `SIGPIPE` signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

`SOCK_DGRAM` and `SOCK_RAW` sockets allow sending of datagrams to correspondents named in send calls. You can also receive datagrams at such a socket with `recv`. Connected `SOCK_DGRAM` sockets can communicate through the `read` and `write` system calls.

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

ACCESS CONTROL

The access depends on the domain and type of service requested, see information about the individual domain for restrictions. However, in general only superuser can use sockets of type `SOCK_RAW`.

RETURN VALUE

The return value is a descriptor referencing the socket.

0..maxfd A file descriptor which references the created socket.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EAFNOSUPPORT	The specified address family is not supported in this version of the system.
EACCES	Permission to create a socket of the specified type and/or protocol is denied.
ESOCKTNOSUPPORT	The specified socket type is not supported in this address family.
EPROTONOSUPPORT	The specified protocol is not supported.
ENFILE	The per-system descriptor table is full.
ENOBUFS	No buffer space is available. The socket cannot be created.
EPROTOTYPE	The protocol type doesn't supply the desired type of service.
ENOSTR	The system is out of STREAMS resources and could not create the protocol stream.

SEE ALSO

accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), recv(2), select(2), send(2), shutdown(2), socketpair(2), inet(3N), unix_ipc(6F).

NAME

`socketpair` – create a pair of connected sockets

SYNOPSIS

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

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

where:

d	Domain of the socket, PF_UNIX
type	Type of service, SOCK_STREAM/SOCK_DGRAM
protocol	Protocol of interest, 0 for default
sv	Buffer in which to return descriptors

DESCRIPTION

The `socketpair` call creates an unnamed pair of connected sockets in the specified domain *d*, of the specified type, and using the optionally specified protocol. The descriptors used in referencing the new sockets are returned in *sv*`[0]` and *sv*`[1]`. The two sockets are indistinguishable.

This call is currently implemented only for the UNIX domain.

ACCESS CONTROL

See related documentation on the domain of interest.

RETURN VALUE

0	Completed successfully.
–1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EMFILE	Too many descriptors are in use by this process.
ENFILE	No per-system file descriptor available.
EAFNOSUPPORT	The specified address family is not supported on this machine.
EPROTONOSUPPORT	The specified *protocol* is not supported on this machine.
EOPNOSUPPORT	The specified *protocol* does not support creation of socket pairs.
EFAULT	The address *sv*`[]` does not specify a valid part of the process address space.
ENOBUFS	No internal buffers available.

SEE ALSO

`read`(2), `write`(2), `inet`(3N), `unix_ipc`(6F).

NAME

stat – get file status

SYNOPSIS

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

int  stat (path, buffer_ptr)
char * path;
struct stat * buffer_ptr;
```

where:

path	Address of a pathname
buffer_ptr	Address of a stat buffer to fill

DESCRIPTION

Stat returns the current attributes of the file named by the pathname pointed to by *path* into the stat buffer at the location specified by *buffer_ptr*. If *path* refers to a symbolic link, file status for the target of the symbolic link is returned.

The interpretation of the file's attributes depends on the file's type (see stat(5)). The subject file must be of type 'ordinary-disk-file', 'directory', 'block-special-file', 'character-special-file', or 'fifo-special-file'.

If stat fails, the contents of the buffer are undefined.

ACCESS CONTROL

Read, write, or execute permission of the named file is not required, but the process must have permission to resolve *path*.

RETURN VALUE

0	The stat operation was successful.
–1	An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EFAULT	*Buffer_ptr* points to an invalid address.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the

process's address space.

SEE ALSO

chmod(2), chown(2), creat(2), dg_mstat(2), fchmod(2), fchown(2), fstat(2), link(2), lstat(2), mknod(2), pipe(2), read(2), time(2), unlink(2), utime(2), utimes(2), write(2), stat(5).

NAME

`statfs` – get information about a mounted file system

SYNOPSIS

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

int  statfs (pathname, statfs_buffer, len, fstype)
char * pathname;
struct statfs * statfs_buffer;
int    len;
int    fstype;
```

where:

pathname	Address of a pathname
statfs_buffer	Where information about the file system is returned
len	Length of the statfs structure
fstype	0 (to return the file system statistics for the file system containing the file named by *pathname*) or nonzero (to return the file system statistics for the file system that resides on the file system device named by *pathname*)

DESCRIPTION

If *fstype* is 0, `statfs` returns information about the mounted file system that contains the file named by *pathname*. Otherwise, `statfs` returns information about the file system residing on the device named by *pathname*. Terminal symbolic links are followed. The statistics returned are:

- The file system block size
- The total number of blocks in the file system
- The number of free blocks in the file system
- The number of free blocks that are available to a non-superuser process
- The number of files that the file system is capable of holding
- The number of free file slots in the file system
- A character string file system identifier

See `stat`(5) for details.

Fields that are undefined for a particular file system are set to −1.

ACCESS CONTROL

None.

RETURN VALUE

0 The file system information was successfully returned.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EFAULT	Some part of the statfs structure pointed to by *statfs_buffer* lies outside of the process's writable address space.
ENOENT	The named file does not exist.

ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.
EINVAL	*Fstype* was nonzero and *pathname* did not name a block special device.

SEE ALSO

`chmod(2)`, `chown(2)`, `creat(2)`, `fchmod(2)`, `fchown(2)`, `fstatfs(2)`, `link(2)`, `mknod(2)`, `pipe(2)`, `read(2)`, `time(2)`, `times(2)`, `ustat(2)`, `write(2)`, `fs(4)`, `statfs(5)`.

NAME

`statvfs` – return information about a file system

SYNOPSIS

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

int  statvfs (const char *pathname, struct statvfs *buffer)
```

where:

pathname The pathname of a file in the file system to be reported on

buffer Address of statvfs buffer where file system information will be returned

DESCRIPTION

Pathname must be that of a file residing in the file system desired for report on. Read, write, or execute permission to the file is not required, but all directories preceeding the file named must be searchable. The information returned about the file system includes:

```
ulong f_bsize;     /* file system block size */
ulong f_frsize;    /* file system fragment size */
ulong f_blocks;    /* total number of blocks of f_frsize
                      contained in the file system */
ulong f_bfree;     /* total number of free blocks */
ulong f_bavail;    /* number of free blocks available to
                      the non-super-user */
ulong f_fsid;      /* file system identifier */
char  f_basetype[FSTYPSZ]; /* null-terminated fs type
                              name */
ulong f_flag;      /* bit mask of flags */
ulong f_namemax;   /* maximum file name length */
char  f_fstr[32];  /* file system specific string */
```

f_basetype contains the file system type name and is null-terminated. The value for the constant FSTYPSZ is defined in the <statvfs.h> file.

The f_flag can return the following:

```
ST_RDONLY      /* a read-only file system */
ST_NOSUID      /* file system does not support the
               setuid or setgid semantics */
```

ACCESS CONTROL

None.

RETURN VALUE

0 The information was successfully returned in the statvfs buffer.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EACCES If the search permission does not exist on a component of the *pathname*.

ELOOP There were too many symbolic links involved in resolving the path.

ENAMETOOLONG	If the pathname given exceeds the maximum path length.
ENOENT	The file name refered to does not exist.
ENOTDIR	A prefix of the pathname component is not a directory.

SEE ALSO

`chmod`(2), `chown`(2), `create`(2), `dup`(2), `fcntl`(2), `link`(2), `mknod`(2), `open`(2), `pipe`(2), `read`(2), `time`(2), `unlink`(2), `ustat`(2), `utime`(2), `write`(2).

NAME

stime – set time

SYNOPSIS

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

int  stime (seconds)
time_t * seconds;
```

where:

seconds Address of an initialized longword interpreted as the new system time

DESCRIPTION

`Stime` sets the system's notion of the current Greenwich time to the value contained in the longword at the location specified by *seconds*.

When the time is successfully changed, a log of the change is sent to the error logger device.

The time value specified is interpreted as Greenwich time expressed in seconds since midnight January 1, 1970.

Setting the system clock may interfere with other timing functions.

ACCESS CONTROL

Only the superuser may set the time of day.

RETURN VALUE

0 Completed successfully.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EPERM Permission to change the system time is denied to the calling process.

EFAULT The *seconds* argument references invalid memory.

SEE ALSO

`settimeofday`(2), `time`(2).

NAME

store_conditional – indivisible compare and swap

SYNOPSIS

```
tb0 0,r0,400
```

DESCRIPTION

Store_conditional is a extended operation (XOP) that indivisibly fetches the value of a user memory location, compares it with a value in a register, and if the proper conditions are met, stores a new value into the memory location.

Input registers are:

r2 Address of 32 bit user memory location to be fetched and added to. This address must be aligned on a 4 byte boundary.

r3 The old value against which the value of the memory location will be compared.

r4 The new value to store into the memory location if the proper conditions are met.

r5 A mask used to compute the conditions that govern whether the new value is stored.

Return registers are:

r1 Unchanged

r2 Unchanged

r3 Unchanged

r4 Unchanged

r5 Unchanged

r6 Undefined

r7 Status: 0 means success (memory location was set to the new value), 1 means some fault occurred when accessing the memory location, 2 means the condition was not met and hence the new value was not stored.

r8 Old value of the memory location

r9 Undefined

r10 through r31
Unchanged

The value of the memory location pointed to by r2 is read. If the value read is equal to the value in r3 in all bit positions for which the corresponding bit in the mask (r5) is set, then the new value (r4) is stored into the memory location. More precisely, the value in the memory location is XORed with r3 and ANDed with r5; if the result is 0, the new value is stored into the memory location. If the result is not 0, the new value is not stored, and error code 2 is returned in r7. If any fault (including a page fault) occurs when accessing the memory location, error code 1 is returned and the memory location is not modified.

The store_conditional XOP executes indivisibly with respect to all other store_conditional operations running on any processor in the system that may be going on simultaneously to the same physical memory location. It does not necessarily execute indivisibly with respect to store_conditional operations to other memory locations, or with respect to other XOPs to the same memory location, or with respect to normal loads and stores or I/O traffic to the memory location.

While the XOP is being executed, the user process will not be descheduled, will not page fault, and will not be terminated. If a fault of any kind (page fault, protection fault, misaligned access fault, for example) occurs when the XOP references user data, the XOP terminates and returns an error. User code is responsible for catching the error, touching the data item so that the fault can be handled, and then retrying the XOP. The execution time of the XOP is charged to user mode, not kernel mode. User profiling ticks that occur while the XOP is in progress are accounted to the instruction following the trap instruction.

`Store_conditional` must be invoked with an assembly language trap instruction. Typically the trap instruction is done from an assembly language routine that is linked with the application and called as a standard subroutine in the high level language in which the application is written.

EXAMPLE

See example at `fetch_and_add`(2).

SEE ALSO

`fetch_and_add`(2).

NAME

swapon – add a swap device for demand paging

SYNOPSIS

`int swapon(char *special);`

where:

special Pathname of the block device to page on

DESCRIPTION

The `swapon()` function makes the block device *special* available to the system to use for paging. The entire device is made available for use for paging; the previous contents of the storage will be overwritten.

ACCESS CONTROL

The effective user id of the calling process must be superuser.

RETURN VALUE

Upon successful completion, `swapon()` returns a value of `0`. Otherwise, it returns the value `-1`, and sets `errno` to indicate an error.

DIAGNOSTICS

Under the following conditions, `swapon()` fails and sets `errno` to:

EPERM if the effective user id of the calling process is not superuser.

ENOSPC if the swap area could not be set up because the system already has the maximum number of paging areas in use.

ENODEV if the swap area could not be set up because its size is bigger than the maximum or smaller than the minimum allowable size for a paging area.

ENOTBLK if the file with the specified pathname is not a block special file.

EBUSY if the given device is already in use.

ENOENT if there is no file with the specified pathname.

ENOENT if a non-terminal component of the specified pathname does not exist.

ENOTDIR if a non-terminal component of the specified pathname was not a directory or symbolic link.

ENAMETOOLONG
if the pathname exceeds the length limit for pathnames.

ENAMETOOLONG
if a component of the pathname exceeds the length limit for filenames.

ELOOP if the number of symbolic links encountered during pathname resolution exceeds the system maximum. A symbolic link cycle is suspected.

EFAULT if the pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

swapon(1M).

NAME

`symlink` – create a symbolic link file

SYNOPSIS

```
#include <unistd.h>

int  symlink (link_contents, link_path)
char * link_contents;
char * link_path;
```

where:

link_contents	Null terminated string to become the symbolic link's contents
link_path	Address of a pathname

DESCRIPTION

`Symlink` creates a symbolic link file named by the pathname pointed to by *link_path* that contains the null-terminated string pointed to by *link_contents*.

Link_contents need not be a valid pathname in order to create the symbolic link. When the symbolic link is resolved as part of a pathname, however, an error will occur if it does not obey all the pathname resolution rules.

Link_contents must be less than MAXPATHLEN bytes long. This restriction is in addition to the size restrictions that apply to every file – the process file size limit, and the system file size limit.

The symbolic link file is entered into the filesystem. The file's attributes are initialized as follows:

- The inode number (st_ino) refers to the per-file database allocated.
- The device number (st_dev) is the same as that of the directory containing the symbolic link file. The represented device (st_rdev) is undefined.
- The number of links (st_nlink) is set to one.
- The file mode (st_mode) is set as follows: The file type is 'symbolic-link-file'. The other mode fields are undefined.
- The user id (st_uid) is set to the effective user id of the calling process. (The user id is needed to support the protection required by readlink.) The group id (st_gid) is undefined.
- The file size (st_size) is set to the number of characters in *link_contents*, exclusive of the terminating null character.
- The time last accessed (st_atime), time last modified (st_mtime), and time of last attribute change (st_ctime) are set to the current time.

Link_path is created in the containing directory and is made to identify the newly created file. An allocation to the directory causes its attributes to change as follows:

- The file size (st_size) is updated if the number of block necessary to hold the directory entries was increased by adding the symbolic link entry.
- The time last modified (st_mtime) and time of last attribute change (st_ctime) are set to the current time.

If `symlink` fails, no changes are made.

ACCESS CONTROL

The calling process must have permission to resolve *link_path*.

The calling process must have write permission to the directory containing the symbolic link to be added.

RETURN VALUE

0 The symbolic link was successfully created.

−1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EEXIST	The symbolic link named by *new_path* exists.
EFAULT	*Link_contents* points outside the allocated address space of the process.
EROFS	The requested symbolic link requires writing in a directory on a file system mounted read-only.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

`ln`(1), `link`(2), `readlink`(2), `unlink`(2), `stat`(5).

NAME

sync – synchronize disk and memory resident file system information

SYNOPSIS

```
#include <unistd.h>

void  sync ()
```

DESCRIPTION

The sync system call causes file system information in memory to be written to the disk.

Activity may continue on the file system device while the sync is being performed, but there are no guarantees about whether changes to files or file system data that occur after sync starts get to disk. Upon return from sync, there is no guarantee that all writes to the disk have completed.

ACCESS CONTROL

None.

RETURN VALUE

None.

DIAGNOSTICS

None.

SEE ALSO

sync(1M), fsync(2).

NAME

sysconf – get configurable system variables

SYNOPSIS

```
#include <unistd.h>
#include <sys/m88kbcs.h>

long sysconf (name)
int  name;
```

where:

name The name of the system variable to be queried

DESCRIPTION

The `sysconf()` function provides a method for the application to determine the current value of a configurable system limit or option (*variable*).

The implementation shall support all of the variables listed in the table "Configurable System Variables" and may support others. The variables in the table come from `<limits.h>` or `<unistd.h>` (or `<time.h>` from the C Standard for {CLK_TCK}), and the symbolic constants, defined in `<unistd.h>`, that are the corresponding values used for *name*.

Configurable System Variables

Variable	*name* Value
{ARG_MAX}	{_SC_ARG_MAX}
{CHILD_MAX}	{_SC_CHILD_MAX}
{CLK_TCK}	{_SC_CLK_TCK}
{NGROUPS_MAX}	{_SC_NGROUPS_MAX}
{OPEN_MAX}	{_SC_OPEN_MAX}
{PAGESIZE}	{_SC_PAGESIZE}
{_POSIX_JOB_CONTROL}	{_SC_JOB_CONTROL}
{_POSIX_SAVED_IDS}	{_SC_SAVED_IDS}
{_POSIX_VERSION}	{_SC_VERSION}

The value of {CLK_TCK} is permitted to be evaluated at run-time by the C Standard (and thus by this standard). The value returned by `sysconf()` for {_SC_CLK_TCK} shall be the same as that returned by {CLK_TCK}.

RETURN VALUE

If *name* is an invalid value, `sysconf()` shall return –1. If the variable corresponding to *name* is not defined on the system, `sysconf()` shall return –1 without changing the value of `errno`.

Otherwise, the `sysconf()` function returns the current variable value on the system. The value returned shall not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's `<limits.h>` or `<unistd.h>`. The value shall not change during the lifetime of the calling process.

DIAGNOSTICS

If any of the following conditions occur, `sysconf` returns –1 and sets `errno` to the corresponding value:

EINVAL The value of the *name* argument is invalid.

FILES

unistd.h
sys/m88kbcs.h

SEE ALSO

pathconf(2).

COPYRIGHTS

STANDARDS

In addition to the configurable system variables listed above, the following variables are defined in <sys/m88kbcs.h>:

_SC_BCS_VERSION: Get version number of 88open BCS to which the system conforms.

_SC_BCS_VENDOR_STAMP:
Get BCS vendor stamp of the system.

_SC_BCS_SYS_ID: Get system hardware's unique system ID number.

_SC_MAXUMEMV: Get maximum user process size, in Kbytes.

_SC_MAXUPROC: Get maximum number of processes per user.

_SC_MAXMSGSZ: Get maximum number of bytes in a message.

_SC_NMSGHDRS: Get maximum number of message headers in system.

_SC_SHMMAXSZ: Get maximum size of a shared memory segment.

_SC_SHMMINSZ: Get minimum size of a shared memory segment.

_SC_SHMSEGS: Get maximum number of attached shared memory segments per process.

_SC_NMSYSSEM: Get total number of semaphores in system.

_SC_MAXSEMVL: Get maximum semaphore value.

_SC_NSEMMAP: Get number of semaphore sets.

_SC_NSEMMSL: Get number of semaphores per set.

_SC_NSHMMNI: Get number of shared memory segments in the system.

_SC_ITIMER_VIRT: Determine whether or not system supports a timer.

_SC_ITIMER_PROF: Determine whether or not system supports a profiling timer.

_SC_TIMER_GRAN: Get granularity (in microseconds) of system's real-time clock.

_SC_PHYSMEM: Get system's physical memory size, in Kbytes.

_SC_AVAILMEM: Get amount of physical memory available to user processes, in Kbytes.

_SC_NICE: Determine whether or not nice() process prioritization is supported on system.

_SC_MEMCTL_UNIT:
Get number of bytes in a *memctl*() memory unit.

_SC_SHMLBA: Get number of bytes used as rounding factor on memory addresses by `shmsys`().

_SC_SVSTREAMS: Determine whether or not system supports System V style STREAMS.

_SC_CPUID: Get the value stored in the M88100 Processor Identification Register.

_SC_NPTYS: Get the number of BCS Networking Supplement stype pseudo-ttys supported.

NAME

`sysfs` – returns information about file system types

SYNOPSIS

```
#include <sys/fstyp.h>
#include <sys/fsid.h>

int  sysfs (int opcode, parameter1, parameter2)
```

where:

opcode — The operation code to get file system information (GETFSIND, GETFSTYP, or GETNFSTYP)

parameter1 — Parameter's existence and use depends on operation.

parameter2 — Parameter's existence and use depends on operation.

DESCRIPTION

`Sysfs` returns information about the file system types configured in the system. The number of arguments accepted by sysfs varies and depends on the *opcode* selected. The recognized *opcodes* and their functions are described below:

GETFSIND — Translates *fsname* (*parameter1*), a null-terminated file system identifier, into a file system type index. *Parameter2* is ignored.

GETFSTYP — Translates *fs_index* (*parameter1*), a file system type index into a null-terminated file system identifier and writes it into the buffer pointed to by *fsname* (*parameter2*). This must be at least the size of FSTYPSZ as defined in `<sys/fstyp.h>`.

GETNFSTYP — Returns the total number of file system types configured with the system. *Parameter1* and *parameter2* are ignored.

ACCESS CONTROL

None.

RETURN VALUE

For GETFSIND:

fs_index or −1
: `Errno` is set to indicate the error if −1. Otherwise, it is the type index of the file system.

For GETFSTYP:

0 — Successful, paramter2 is set to the file system name.

−1 — An error occurred. Errno is set appropriately.

For GETNFSTYP:

Number of registered file systems
: The return value indicates the number of configured file systems.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EINVAL — *Fsname* points to an invalid file system identifier; fs_index is zero, or invalid; the *opcode* is invalid.

EFAULT — A pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

`mount`(2), `nfsmount`(2), `fs`(4).

NAME

`sysinfo` – get and set system information strings

SYNOPSIS

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

long sysinfo (int command, char *buf, long count);
```

where:

command Specifies a particular operation for `sysinfo` to perform.

buf A pointer to a buffer where system information will be written. If *command* specifies a set operation, then *buf* will point to a string that `sysinfo` will use to set a system variable.

count The length in bytes of the buffer pointed to by *buf*.

DESCRIPTION

`Sysinfo` copies information relating to the system on which the process is executing into the buffer pointed to by *buf*; `sysinfo` can also set certain information as specified by *commands*.

The POSIX P1003.1 interface `sysconf` [see `sysconf(2)`] provides a similar class of configuration information, but returns an integer rather than a string.

The commands you can specify are as follows:

SI_SYSNAME
: Copy into the array pointed to by *buf* the string that would be returned by `uname` [see `uname(2)`] in the *sysname* field. This is the name of the implementation of the operating system, e.g., *dgux*.

SI_HOSTNAME
: Copy into the array pointed to by *buf* a string that names the present host machine. This is the string that would be returned by *uname* [see *uname*(2)] in the *nodename* field.

The *hostname* is the name of this machine as a node in some network; different networks may have different names for the node, but presenting the nodename to the appropriate network Directory or name-to-address mapping service should produce a transport end point address. The name may not be fully qualified.

Internet host names may be up to 256 bytes in length (plus the terminating null).

SI_SET_HOSTNAME
: Copy the null-terminated contents of the array pointed to by *buf* into the string maintained by the kernel whose value will be returned by succeeding calls to `sysinfo` with the command `SI_HOSTNAME`. This command requires that the effective-user-id be super-user.

SI_RELEASE
: Copy into the array pointed to by *buf* the string that would be returned by `uname` [see `uname(2)`] in the *release* field. A typical value might be *5.4*.

SI_VERSION
: Copy into the array pointed to by *buf* the string that would be returned by `uname` [see `uname(2)`] in the *version* field. The syntax and semantics of this string are defined by the system provider.

SI_MACHINE
: Copy into the array pointed to by *buf* the string that would be returned by `uname` [see `uname(2)`] in the *machine* field,*e.g., AViiON*.

SI_ARCHITECTURE
: Copy into the array pointed to by *buf* a string describing the instruction set architecture of the current system, e.g., *mc88100*. These names may not match predefined names in the C language compilation system.

SI_HW_PROVIDER
: Copies the name of the hardware manufacturer into the array pointed to by *buf*, e.g., *Data General*.

SI_HW_SERIAL
: Copy into the array pointed to by *buf* a string which is the ASCII representation of the unique, hardware-specific serial number of the physical machine on which the system call is executed.

SI_SRPC_DOMAIN
: Copies the Secure Remote Procedure Call domain name into the array pointed to by *buf*.

SI_SET_SRPC_DOMAIN
: Set the string to be returned by `sysinfo` with the SI_SRPC_DOMAIN command to the value contained in the array pointed to by *buf*. This command requires that the effective-user-id be super-user.

RETURN VALUE

Upon successful completion, the return value indicates the buffer size in bytes required to hold the complete string value and the terminating null character. If this value is no greater than the value passed in *count*, the entire string was copied; if this value is greater than *count*, the string copied into *buf* has been truncated to *count−1* bytes plus a terminating null character.

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

DIAGNOSTICS

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

EPERM The process does not have appropriate privilege for a SET commands.

EINVAL *buf* does not point to a valid address, or the data for a SET command exceeds the limits established by the implementation.

A good starting guess for *count* is 257, which is likely to cover all strings returned by this interface in typical installations.

SEE ALSO

`uname(2)`, `sysconf(2)`, `gethostname(2)`, `gethostid(2)`.

NOTE

For many of the system information variables, no programmatic interface exists that allows a user set their values. Such strings are settable only by the system administrator modifying entries in the `master.d` directory.

NAME

`time` – get system time

SYNOPSIS

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

time_t time (tloc)
time_t * tloc;
```

where:

tloc NULL or address of a `time_t` to be set to the current system time

DESCRIPTION

`Time` returns the system's notion of the current Greenwich time as the system call's return value.

If *tloc* is not NULL, the current time is also stored in the (time_t) at the location specified by *tloc*.

The time value returned is Greenwich time expressed in seconds since midnight January 1, 1970.

ACCESS CONTROL

None.

RETURN VALUE

current time Completed successfully.

-1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EFAULT Time will fail if *tloc* points to an illegal address.

SEE ALSO

`date`(1), `stime`(2).

NAME

`times` – get process and child process times

SYNOPSIS

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

clock_t   times (buffer)
struct tms   * buffer;
```

where:

buffer The address of a structure into which the times are to be put

DESCRIPTION

The `times` system call fills in the structure pointed to by *buffer* with time accounting information for the calling process and each of its terminated child processes for which it has executed a `wait`.

All times are defined in units of 1/{CLK_TCK} of a second. See sysconf(2).

The value of *tm_utime* is the CPU time used while executing instructions in the user space of the calling process.

The value of *tm_stime* is the CPU time used by the system on behalf of the calling process.

The value of *tms_cutime* is the sum of the *tms_utime* and *tms_cutime* of the child processes.

The value of *tms_sutime* is the sum of the *tms_stime* and *tms_cstime* of the child processes.

ACCESS CONTROL

The argument *buffer* must point to an area of the calling process's address space that is valid and has write access.

RETURN VALUE

Upon successful completion, times returns the elapsed real-time, in [CLK_TCK]ths of a second, since an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of times to another. Hence, a single value returned from this call is not meaningful; only the difference between values returned at different times is meaningful.

If an error occurs, −1 is returned and `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EFAULT *Buffer* points to an illegal address.

SEE ALSO

`time`(1), `exec`(2), `fork`(2), `time`(2), `wait`(2).

NAME

`truncate` – truncate a file to a specified length

SYNOPSIS

```
#include <unistd.h>

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

where:

path	Address of a pathname
length	Maximum length of file after truncation

DESCRIPTION

`Truncate` causes the file named by *path* to be truncated to at most *length* bytes in size. If *path* refers to a symbolic link, the target of the symbolic link is truncated.

The subject file must reside on a file system device mounted read-write. Also, it must not be a directory. If mandatory locking is enabled on the file, `truncate` waits until all locks on the file are cleared.

If an error occurs, no changes occur. Otherwise, the subject file is changed with the following consequences:

- For files of type 'ordinary-disk-file', if the file's size is greater than *length* bytes, it is truncated to that length, and the file's size is updated. If the file's size is less than *length* bytes, the file is lenghtened by appending null bytes and the file's size is updated.
- If file is not of type 'ordinary-disk-file', neither its contents nor its size are altered.
- The 'time-last-modified' and 'time-last-changed' attributes are set to the current time. These attributes are changed even if there is no change to the file's contents.

ACCESS CONTROL

The calling process must have permission to resolve *path*.

The calling process must have write access to the file.

RETURN VALUE

0	The file was successfully truncated.
−1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EACCES	Write permission is denied for the named file.
EISDIR	The named file is a directory.
EROFS	The named file resides on a file system device mounted read-only.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.

ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.
EINTR	The `truncate` system call was interrupted while waiting for a mandatory record lock to clear.

SEE ALSO

`creat`(2), `ftruncate`(2), `open`(2).

STANDARDS

When using `m88kbcs` as the Software Development Environment target, the `truncate` function will be emulated using BCS system calls. Since this emulation uses the open system call, a failure will occur if all file descriptors are in use. In this case, errno will be set to EMFILE. Also, since this is an emulation requiring several BCS system calls, a slight performance degradation may be noticed in comparison to using `truncate` in `/lib/libc.a`.

NAME

uadmin – administrative control

SYNOPSIS

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

int uadmin(int cmd, int fcn, int mdep);
```

where:

cmd A_SHUTDOWN or A_REBOOT

fcn AD_HALT, AD_BOOT, or AD_IBOOT

mdep This argument is provided for machine-dependent use and is not defined here.

DESCRIPTION

Uadmin provides control for basic administrative functions. This system call is tightly coupled to the system administrative procedures and is not intended for general use.

As specified by *cmd*, the following commands are available:

A_SHUTDOWN The system is shut down. All user processes are killed and the buffer cache is flushed. The action to be taken after the system has been shut down is specified by *fcn*. The valid functions are:

- AD_HALT Halt the processor(s).
- AD_BOOT Reboot the system, using the same boot flags as the last time it was booted.
- AD_IBOOT The same as AD_HALT. Control is returned to the system control monitor where the user can specify a new boot path.

A_REBOOT The system stops immediately without any further processing. The action to be taken next is determined by the value of *fcn*.

ACCESS CONTROL

Only the super user may halt the system processor(s).

RETURN VALUE

If successful, this call never returns. Otherwise, a –1 is returned and errno is set to indicate the error.

DIAGNOSTICS

EPERM The effective user ID is not super-user.

SEE ALSO

dg_syscntl(2).

NAME

`ulimit` – get and set user limits

SYNOPSIS

```
#include <sys/ulimit.h>
long ulimit (cmd, newlimit)
int  cmd;
long newlimit;
```

where:

cmd An integer, 0 – 4, specifying which of several user limit-related operations to perform

newlimit An argument to the user limit operation, the specific meaning depending upon the *cmd* argument

DESCRIPTION

The `ulimit` system call controls various per-process limits. The *cmd* argument specifies which of several operations to perform as described below:

GET_ULIMIT or UL_GETFSIZE
: Get the calling process's file size limit. The file size limit is the maximum logical offset within a file at which the process can perform a write operation. The limit is in units of 512-byte blocks. The *newlimit* argument is ignored and need not be present. This option is the same as the hard RLIMIT_FSIZE in `getrlimit`.

SET_ULIMIT or UL_SETFSIZE
: Set the file size limit of the process to *newlimit*. A process may not increase its file size limit unless it has an effective-user-id of 0 (that is, is super-user). *Newlimit* may be any positive or negative integer. This option is the same as the hard RLIMIT_FSIZE in `setrlimit` with the soft RLIMIT_FSIZE set to RLIM_INFINITY.

GET_BREAK
: Get the maximum possible break value for the calling process. The *newlimit* argument is ignored and need not be present.

GET_MAX_OPEN
: Get the maximum number of open files allowed per process.

ACCESS CONTROL

The following access restrictions apply, depending on the value of *cmd*:

GET_ULIMIT or UL_GETFSIZE
: None.

SET_ULIMIT or UL_SETFSIZE
: If *newlimit* is greater than the current value of the file size limit, the effective-user-id of the calling process must be 0 for the call to succeed. Otherwise, the limit is unchanged and an EPERM error is returned.

GET_BREAK
: None.

GET_MAX_OPEN
: None.

RETURN VALUE

If *cmd* has the value GET_ULIMIT or UL_GETFSIZE, the return value is as follows:

0..FILESIZE The return value is always the current value of the calling process's file size limit.

If *cmd* has the value SET_ULIMIT or UL_SETFSIZE, the return value is as follows:

0..FILESIZE Successful completion. The new file size limit is returned.

−1 An error occurred. `errno` is set to indicate the error.

If *cmd* has the value GET_BREAK, the return value is as follows:

0..MAXBRK The return value is always the calling process's maximum possible break value.

If *cmd* has the value GET_MAX_OPEN, the return value is as follows:

NOFILE The return value is always NOFILE as defined in `param.h`.

If *cmd* is anything other than the above values, −1 is returned and `errno` is set to EINVAL.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EPERM The calling process is trying to increase its file size limit and does not have an effective-user-id of 0.

EINVAL The value of *cmd* was not one of the valid commands listed above.

SEE ALSO

`brk`(2), `getrlimit`(2), `setrlimit`(2), `write`(2).

NAME

umask – set and get file creation mask

SYNOPSIS

```
mode_t  umask (creation_mask)
int     creation_mask;
```

where:

creation_mask File mode creation mask

DESCRIPTION

Umask sets the process's file mode creation mask to the low-order 9 bits of *creation_mask* and returns the previous value of the mask. Those bits other than the low-order 9 in *creation_mask* are reserved, and in the return value are undefined. See creat(2) for details of how this mask is used.

ACCESS CONTROL

None.

RETURN VALUE

0..07777 Previous mask value.

DIAGNOSTICS

None.

SEE ALSO

mkdir(1), sh(1), umask(1), chmod(2), creat(2), mknod(2), open(2).

NAME

`umount` – remove a file system device

SYNOPSIS

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

int  umount (special)
char * special;
```

where:

special Address of a pathname

DESCRIPTION

`Umount` removes the file system device identified by *special* or mounted on the file *special* from the set of active file system devices with the following consequences:

- The filename store contained on *special* is removed from the system filename store. Thus, all files contained on *special* can no longer be named.
- The filesystem contained on *special* is removed from the system flat file store. Thus, all files contained on *special* can no longer be accessed.
- None of the files on *special* may be open. No process may have its current working directory on *special*.
- The filename store contained on *special* cannot contain a mount point of any other file system device at the time of the call to `umount`.
- *Special* must have previously been the subject of a successful mount operation. If `umount` is successful, the sub-tree over which *special* was mounted reappears in the system file name store. These files can now be named.
- If *special* refers to a named stream and there are no other references to the stream, the stream is closed and its resources deallocated.

If an error occurs, no changes are made.

ACCESS CONTROL

To unmount a dg/ux or nfs file system, the calling process's effective user id must be the superuser. To unmount a namefs file system, the calling process's effective user id must be the superuser or the owner of *special*.

RETURN VALUE

0	*Special* was successfully unmounted.
–1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBUSY	There are still processes accessing file system objects on *special*.
EBUSY	A file contained on *special* is the mount point of another file system device.
EINVAL	*Special* is not mounted.
ENOENT	The named file does not exist.
ENOTBLK	*Special* is not a block special file.
ENXIO	The device associated with *special* does not exist.

EIO	I/O error when flushing file system information.
EPERM	Permission to unmount the file system device is denied to the calling process.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

mount(1M), dg_mount(2), mount(2), fs(4).

NAME

uname, nuname – get name of current UNIX system

SYNOPSIS

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

int    uname (name)
struct utsname * name;

int    nuname (name)
struct utsname * name;
```

where:

name Address of a structure to be filled with the current system name

DESCRIPTION

uname and its synonym nuname store information identifying the current UNIX system in the structure pointed to by *name*. This information is set during the system generation procedure and may be meaningful to other software. The utsname structure is defined in the include file <sys/utsname.h>. See <sys/utsname.h> for a description of the fields.

ACCESS CONTROL

None.

RETURN VALUE

0 The operation was successful.

–1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EFAULT uname and nuname will fail if *name* points to an invalid address.

SEE ALSO

uname(1) in the *User's Reference for the DG/UX System*, hostname(1C), sethostname(2)

NOTES

The command hostname(1C) and the system call sethostname(2) modify the system's nodename, and thus change the value returned in the nodename field of the utsname structure.

NAME

`unlink` – remove a directory entry

SYNOPSIS

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

where:

path Address of a pathname

DESCRIPTION

`Unlink` removes the directory entry named by the pathname pointed to by *path* from the filename store. A symbolic link occurring at the end of *path* will not be followed. The named file must reside on a file system device mounted read-write.

The subject file must be of type 'ordinary-disk-file', 'block-special-file', 'character-special-file', 'fifo-special-file', 'socket', or 'symbolic-link'.

It is an error to attempt an `unlink` call on a directory or control point directory type file.

Removing a reference to a file in the filename store has the following consequences:

- The subject file's link count attribute is decremented.
- The subject file's 'time-last-attribute-changed' attribute is set to the current time.
- If the subject file has no more links in the filename store, then on the release of the last reference, the file will removed from the flat file store. Thus, `unlink` deletes inactive files.
- If the subject file has no more links in the filename store but is still open, then the file is removed from the filesystem when it is closed for the last time.

If `unlink` fails, no changes are made to the named file.

ACCESS CONTROL

The calling process must have permission to resolve *path*.

The calling process must have write permission to the directory containing the entry to be removed.

RETURN VALUE

0 The filename was successfully removed.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EACCES	Permission to modify the directory containing the entry to be removed is denied to the calling process.
EBUSY	The named file is the mount point of a file system device.
EPERM	The named file is a directory.
EROFS	The named file is contained on a read-only file system device.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.

ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.
EFAULT	The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

`rm`(1), `close`(2), `link`(2), `open`(2).

NAME

ustat – get file system device statistics

SYNOPSIS

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

int     ustat (device, ustat_buffer)
dev_t   device;
struct ustat * ustat_buffer;
```

where:

device	Device number of a mounted file system device
ustat_buffer	Where the file system statistics are returned

DESCRIPTION

`Ustat` returns information about a mounted file system device. *Device* is a device number identifying a device containing a mounted file system. Status information concerning the file system device contained on *device* is returned in the location pointed to by *ustat_buffer*.

If an error occurs, the contents of *ustat_buffer* are undefined.

ACCESS CONTROL

None.

RETURN VALUE

0	The file system device status information was successfully returned.
–1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EINVAL	*Device* is not the device number of an active file system device.
EFAULT	Some part of the ustat structure pointed to by *ustat_buffer* lies outside of the process's writable address space.
EINTR	Interrupted `ustat` call.

SEE ALSO

`fstat`(2), `stat`(2), `fs`(4), `ustat`(5).

NAME

utime – set file access and modification times

SYNOPSIS

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

int  utime (path, times)
char * path;
struct utimbuf *times;
```

where:

path Address of a pathname

times NULL or address of an initialized structure giving the access and modification times

DESCRIPTION

Path points to a pathname naming a file, which must reside on a file system device mounted read-write. If *path* refers to a symbolic link, the target of the symbolic link is affected. `Utime` sets the 'time-last-accessed' and 'time-last-modified' attributes of the subject file. If *times* is NULL, 'time-last-accessed' and 'time-last-modified' are set to the current time. Otherwise, the 'time-last-accessed' is set to `(*times).actime` and 'time-last-modified' is set to `(*times).modtime`.

If `utime` fails, the file is left unchanged. Otherwise, the 'time-last-changed' attribute of the subject file is set to the current time.

ACCESS CONTROL

The calling process must have permission to resolve *path*.

If *times* is NULL, either the calling process must have write permission to the subject file or the calling process's effective user id must be equal to the user id of the subject file.

Otherwise, the calling process's effective user id must be superuser or the user id of the subject file.

RETURN VALUE

0 The file's access and modification times were changed successfully.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EACCES	Permission to set the access and modification times to the current time is denied to the calling process.
EFAULT	*Times* is not NULL and some part of the `utimbuf` structure pointed to by *times* lies outside the process's readable address space.
EPERM	Permission to set the access and modification times to an arbitrary value is denied to the calling process.
EROFS	The file system device containing the subject file is mounted read-only.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.

ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle is suspected.
EPERM	The pathname contains a character not in the allowed character set.

SEE ALSO

touch(1), dg_mstat(2), lstat(2), stat(2), ustat(2), utimes(2), stat(5).

NAME

`utimes` – set file access and modification times

SYNOPSIS

```
#include &lt;sys/time.h>

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

where:

path Address of a pathname naming a file, which must reside on a file system device mounted read-write

times Address of an initialized array of two time values giving the access and modification times

DESCRIPTION

`Utimes` sets the 'time-last-accessed' and 'time-last-modified' attributes of the subject file to *times*[0] and *times*[1] respectively. If *path* refers to a symbolic link, the target of the symbolic link is affected.

If `utimes` fails, the file is left unchanged. Otherwise, the 'time-last-changed' attribute of the subject file is set to the current time.

ACCESS CONTROL

The calling process must have permission to resolve *path*.

The calling process's effective user id must be superuser or the user id of the subject file.

RETURN VALUE

0 The file's access and modification times were changed successfully.

–1 An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EFAULT	Some part of the array pointed to by *times* lies outside the process's readable address space.
EPERM	Permission to set the access and modification times to an arbitrary value is denied to the calling process.
EROFS	The file system device containing the subject file is mounted read-only.
ENOENT	The file the pathname resolved to does not exist.
ENOENT	A non-terminal component of the pathname does not exist.
ENOTDIR	A non-terminal component of the pathname was not a directory or symbolic link.
ENAMETOOLONG	The pathname exceeds the length limit for pathnames.
ENAMETOOLONG	A component of the pathname exceeds the length limit for filenames.
ENOMEM	There are not enough system resources to resolve the pathname or to expand a symbolic link.
ELOOP	The number of symbolic links encountered during pathname resolution exceeded MAXSYMLINKS. A symbolic link cycle

is suspected.

EPERM The pathname contains a character not in the allowed character set.

EFAULT The pathname does not completely reside in the process's address space or the pathname does not terminate in the process's address space.

SEE ALSO

dg_mstat(2), lstat(2), stat(2), ustat(2), utime(2), stat(5).

NAME

`vfork` – spawn new process in a virtual memory efficient way

SYNOPSIS

```
#include <unistd.h>

int  vfork ()
```

DESCRIPTION

`vfork` creates a new process in the same way that `fork(2)` does except that the new process (the child) shares the address space of the parent rather than being given his own address space that is a copy of the parent's. The `vfork` call does not return in the parent process until the child does an `exec`, an `_exit`, or terminates abnormally. The `vfork` call does return in the child process, whereupon it is expected the child will call `exec` very soon.

`vfork` can normally be used just like `fork`, except after the `vfork` call the child must be careful about modifying the user address space and any per-process state, since the changes will be reflected in the parent when he continues. It does not work, for example, for the child process to return from the procedure which called `vfork` because the parent would return to a no-longer-existent stack frame.

If the following process attributes are changed by the child, those changes will be visible to the parent:

- The shared memory segments (see `shmat` and `shmdt`).
- The unshared data segment as a result of changing the break value (see `brk` and `sbrk`).
- The text or data segment locks (see `plock`).

ACCESS CONTROL

No access checking is performed.

RETURN VALUE

Upon successful completion, `vfork` returns a value of 0 to the child process and (later) 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.

DIAGNOSTICS

`vfork` 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.
EAGAIN	The calling process is not a superuser and there already exists cf_pm_max_processes_per_real_user_id processes with the same real user id as the calling process.
ENOMEM	The process requires more space than the system is able to supply.

SEE ALSO

`exec(2)`, `fork(2)`, `nice(2)`, `plock(2)`, `ptrace(2)`, `semop(2)`, `signal(2)`, `sigset(2)`, `times(2)`, `ulimit(2)`, `umask(2)`, `wait(2)`.

NOTES

To avoid a possible deadlock, child processes in the middle of a `vfork` are never sent SIGTTOU or SIGTTIN signals; rather, output or `ioctls` are allowed and input attempts result in an end-of-file indication.

STANDARDS

When using `m88kbcs` as the Software Development Environment target, the `vfork` function will be an incomplete emulation of Berkeley semantics. This emulation does not support the virtual fork capability but is simply a call to `fork`.

NAME

`vhangup` – virtually hang up the current control terminal

SYNOPSIS

```
void  vhangup ()
```

DESCRIPTION

This function provides capabilities that are inherently implementation dependent. It may change or cease to exist in the future.

`vhangup` revokes read and write access to the calling process's controlling terminal for all processes (including the calling process). Further attempts to access this terminal will cause I/O errors (EBADF). If the subject terminal has a process group associated with it, a hangup signal (SIGHUP) is sent to that process group.

ACCESS CONTROL

The calling process's effective user id must be superuser.

RETURN VALUE

None.

DIAGNOSTICS

None.

SEE ALSO

`init`(1M).

NAME

`wait`, `waitpid` – wait for process termination

SYNOPSIS

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

pid_t wait(stat_loc)
int *stat_loc;

pid_t waitpid (pid, stat_loc, options)
pid_t pid;
int *stat_loc;
int options;
```

where:.

pid	A process identifier
stat_loc	A location for returning a process status
options	0 or a positive integer (see "Option Flags" below)

DESCRIPTION

The `wait()` and `waitpid()` functions allow the calling process to obtain status information pertaining to one of its child processes. Various options permit status information to be obtained for child processes that have terminated or stopped. If status information is available for two or more child processes, the order in which their status is reported is unspecified.

The `wait()` function shall suspend execution of the calling process until status information for one of its terminated child processes is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If status information is available prior to the call to `wait()`, return shall be immediate.

The `waitpid()` function shall behave identically to the `wait()` function, if the *pid* argument has a value of −1 and the *options* argument has a value of zero. Otherwise, its behavior shall be modified by the values of the *pid* and *options* arguments.

The *pid* argument specifies a set of child processes for which status is requested. The `waitpid()` function shall only return the status of a child process from this set.

(1) If *pid* is equal to −1, status is requested for any child process. In this respect, `waitpid()` is then equivalent to `wait()`.

(2) If *pid* is greater than zero, it specifies the process ID of a single child process for which status is requested.

(3) If *pid* is equal to zero, status is requested for any child process whose process group ID is equal to that of the calling process.

(4) If *pid* is less than −1, status is requested for any child process whose process group ID is equal to the absolute value of *pid*.

Option Flags

The *options* argument is constructed from the bitwise inclusive OR of zero or more of the following flags, defined in the header `<sys/wait.h>`:

WUNTRACED If the implementation supports job control, the status of any child processes specified by *pid* that are stopped, and whose status has not yet been reported since they stopped, shall also be reported to the requesting process.

WCONTINUED Also report the status of any continued child process specified by *pid* whose status has not been reported since it continued.

WNOHANG The `waitpid()` function shall not suspend execution of the calling process if status is not immediately available for one of the child processes specified by *pid*.

WNOWAIT Keep the process whose status is returned in *stat_loc* in a waitable state. The process may be waited for again with indentical results.

If `wait()` or `waitpid()` return because the status of a child process is available, these functions shall return a value equal to the process ID of the child process. In this case, if the value of the argument *stat_loc* is not NULL, information shall be stored in the location pointed to by *stat_loc*. If and only if the status returned is from a terminated child process that returned a value of zero from *main* or passed a value of zero as the *status* argument to `_exit()` or `exit()`, the value stored at the location pointed to by *stat_loc* shall be zero. Regardless of its value, this information may be interpreted using the following macros, which are defined in `<sys/wait.h>` and evaluate to integral expressions; the *stat_val* argument is the integer value pointed to by *stat_loc*.

WIFEXITED(*stat_val*)
: Evaluates to a non-zero value if status was returned for a child process that terminated normally.

WEXITSTATUS(*stat_val*)
: If the value of WIFEXITED(*stat_val*) is non-zero, this macro evaluates to the low-order 8 bits of the *status* argument that the child process passed to `_exit()` or `exit()`, or the value the child process returned from *main*.

WIFSIGNALED(*stat_val*)
: Evaluates to a non-zero value if status was returned for a child process that terminated due to the receipt of a signal that was not caught (see `<signal.h>`).

WTERMSIG(*stat_val*)
: If the value of WIFSIGNALED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED(*stat_val*)
: Evaluatès to a non-zero value if status was returned for a child process that is currently stopped.

WSTOPSIG(*stat_val*)
: If the value of WIFSTOPPED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

WIFCONTINUED(*stat_val*)
: Evaluates to a non-zero value if status was returned for a child process that has continued.

If the information stored at the location pointed to by *stat_loc* was stored there by a call to the `waitpid()` function that specified the WUNTRACED flag, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), and WIFSTOPPED(**stat_loc*) shall evaluate to a non-zero value. If the information stored at the location pointed to by *stat_loc* was stored there by a call to the `waitpid()` function that did not specify the WUNTRACED flag or by a call to the `wait()` function, exactly one of the macros WIFEXITED(**stat_loc*) and WIFSIGNALED(**stat_loc*) shall evaluate to a non-zero value.

An implementation may define additional circumstances under which `wait()` or `waitpid()` reports status. This shall not occur unless the calling process or one of its child processes explicitly makes use of a nonstandard extension. In these cases the interpretation of the reported status is implementation-defined.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes shall be assigned a new parent process ID corresponding to an implementation-defined system process.

RETURN VALUE

If the `wait()` or `waitpid()` functions return because the status of a child process is available, these functions shall return a value equal to the process ID of the child process for which status is reported. If the `wait()` or `waitpid()` functions return due to the delivery of a signal to the calling process, a value of −1 shall be returned and `errno` shall be set to EINTR. If the `waitpid()` function was invoked with WNOHANG set in *options*, it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, a value of zero shall be returned. Otherwise, a value of −1 shall be returned, and `errno` shall be set to indicate the error.

DIAGNOSTICS

If any of the following conditions occur, the `wait()` function shall return −1 and set `errno` to the corresponding value:

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

EINTR The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

If any of the following conditions occur, the `waitpid()` function shall return −1 and set `errno` to the corresponding value:

ECHILD The process or process group specified by *pid* does not exist or is not a child of the calling process.

EINTR The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

EINVAL The value of the *options* argument is not valid.

SEE ALSO

`_exit(2)`, `fork(2)`, `pause(2)`, `times(2)`, `/usr/include/signal.h`.

COPYRIGHT

STANDARDS

`Wait` and `waitpid` report status if a child process that is being traced [by `ptrace(2)` or `dg_xtrace(2)`] stops. In this case, the 8 low order bits of the wait status will contain the octal value 0177 and the 8 high order bits will contain the value of the signal that stopped the child process.

If the parent process terminates without waiting for all of its child processes to terminate, the remaining child processes are assigned a new parent process ID of 1, which is the PID of the special system initialization process.

NAME

wait3 – wait for child process to stop or terminate

SYNOPSIS

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

int    wait3 (wait_status, options, rusage)
union  wait  * wait_status;
int    options;
struct rusage * rusage;
```

where:

wait_status	NULL or address of a status word
options	Modifications to the action of wait3
rusage	NULL or address of a resource usage structure

DESCRIPTION

The wait3 system call suspends the calling process until a child process stops or terminates, then reports the identity, status, and resource usage of the child process to the calling process. If more than one child process has stopped or terminated, the manner in which one is chosen is undefined.

- A process stops when it is being traced (see ptrace) and either hits a break point or receives a signal that is set to be caught.
- A process terminates when it calls exit or when it receives a signal that causes it to terminate.
- A process that has terminated but whose status has not been reported on by wait may consume system resources. The wait operation cleans up the terminated process and recovers remaining system resources.

wait3 returns the PID of the child process.

The status of the child process is optionally obtained by the *wait_status* parameter. If *wait_status* is NULL, status information is not returned. Otherwise, 16 bits of status information are stored in the low-order 16 bits of the location pointed to by *wait_status*. *wait_status* can be used to determine the reason for the child process' termination.

The following macros are provided in sys/wait.h for use in interpreting *wait_status*. When using these macros with wait3, the program must define _BSD_WAIT_FLAVOR either in the executable or at compile time.

WIFSTOPPED(**wait_status*)
: Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

WIFSIGNALED(**wait_status*)
: Evaluates to a non-zero value if status was returned for a child process that terminated due to the receipt of a signal that was not

WIFEXITED(**wait_status*)
: Evaluates to a non-zero value if status was returned for a child process that terminated normally.

A summary of the resources used by a child process that has terminated is obtained by the *rusage* parameter. If *rusage* is NULL, a summary is not returned. Otherwise, summary information is stored in the location pointed to by *rusage*.

The following statement:

```
wait(status)
```

is identical to this one:

```
wait3(status, 0, NULL)
```

However, the action taken by `wait3` may be modified by setting bits in the *options* parameter as follows.

- The WNOHANG bit specifies that the calling process should not be suspended by `wait3`.
- A process that is not being traced may be stopped by the SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signals. The WUNTRACED bit specifies that the status of all stopped child processes should be reported, not just those being traced.

If a parent process terminates without waiting for its child processes to terminate, a special system process inherits the child processes.

If an error occurs, `wait3` will not clean-up any process and the values of *status* and *rusage* are undefined.

If, while waiting for a child to terminate or stop, the process receives a signal that causes it to invoke a handler, `wait3` will be restarted if the handler was established using `sigvec` without the SV_INTERRUPT flag or `sigaction` with the SA_RESTART flag. See `sigvec(2)` and `sigaction(2)`.

ACCESS CONTROL

None.

RETURN VALUE

child-process-id	Completed successfully.
0	The WNOHANG bit in *options* was set, and the calling process would otherwise have been suspended by `wait3`.
−1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

ECHILD	The calling process has no child processes. This condition implies that the calling process was not suspended by `wait3`.
EFAULT	The *status* or *rusage* arguments point to an illegal address. This condition implies that the calling process was not suspended by `wait3`.

SEE ALSO

`exec(2)`, `_exit(2)`, `fork(2)`, `ptrace(2)`, `sigpause(2)`, `sigvec(2)`, `sigaction(2)`, `wait(2)`, `wait4(2)`, `exit(3C)`.

STANDARDS

When using `m88kbcs` as the Software Development Environment target, the `wait3` function will be an incomplete emulation of Berkeley semantics. Since we are using BCS system calls, resource usage information is not available. If *rusage* is non-NULL, `wait3` will fail with `errno` set to EINVAL.

NAME

`wait4` – wait for the specified child process to stop or terminate

SYNOPSIS

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

int     wait4 (child_pid, wait_status, options, rusage)
int     child_pid;
union   wait    * wait_status;
int     options;
struct rusage * rusage;
```

where:

child_pid	The process id for the child process that we are waiting on
wait_status	NULL or address of a status word
options	Modifications to the action of `wait3`
rusage	NULL or address of a resource usage structure

DESCRIPTION

`Wait4` suspends the calling process until the specified child process stops or terminates, then reports the identity, status, and resource usage of the child process to the calling process.

- A process "stops" when it is being traced (see `ptrace`) and either hits a break point or receives a signal that is set to be caught.
- A process "terminates" when it calls exit either directly or due to the receipt of a signal that causes the process to terminate.
- A process that has terminated, but whose status has not been reported on by `wait` may consume system resources. The `wait` operation "cleans-up" the terminated process and recovers remaining system resources.

The status of the child process is optionally obtained by the *wait_status* parameter. If *wait_status* is NULL, status information is not returned. Otherwise, 16 bits of status information are stored in the low-order 16 bits of the location pointed to by *wait_status*. *wait_status* can be used to determine the reason for the child process' termination.

The following macros are provided in `sys/wait.h` for use in interpreting *wait_status*. When using these macros with `wait4`, the program must define _BSD_WAIT_FLAVOR either in the executable or at compile time.

WIFSTOPPED(**wait_status*)
: Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

WIFSIGNALED(**wait_status*)
: Evaluates to a non-zero value if status was returned for a child process that terminated due to the receipt of a signal that was not

WIFEXITED(**wait_status*)
: Evaluates to a non-zero value if status was returned for a child process that terminated normally.

A summary of the resources used by a child process that has terminated is obtained by the *rusage* parameter. If *rusage* is NULL, a summary is not returned. Otherwise, summary

A summary of the resources used by a child process that has terminated is obtained by the *rusage* parameter. If *rusage* is NULL, a summary is not returned. Otherwise, summary information is stored in the location pointed to by *rusage*.

`Wait4`(2) is identical to `wait3`(2) except that it waits only on one specified child. Siblings of the specified children are ignored. If the caller specifies *child_pid* as zero, `wait4` behaves identically to `wait3`.

The action taken by `wait4` may be modified by setting bits in the *options* parameter as follows.

- The WNOHANG bit specifies that the calling process should not be suspended by `wait4`.
- A process that is not being traced may be stopped by the SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signals. The WUNTRACED bit specifies that the status of all stopped child processes should be reported, not just those being traced.

If a parent process terminates without waiting for its child processes to terminate, a special system process inherits the child processes.

If an error occurs, wait4 will not clean-up any process and the values of *status* and *rusage* are undefined.

If, while waiting for a child to terminate or stop, the process receives a signal that causes it to invoke a handler, `wait4` will be restarted if the handler was established using `sigvec` without the SV_INTERRUPT flag or `sigaction` with the SA_RESTART flag. See `sigvec`(2) and `sigaction`(2).

ACCESS CONTROL

None.

RETURN VALUE

child-process-id	Completed successfully.
0	The WNOHANG bit in *options* was set and the calling process would otherwise have been suspended by `wait3`.
−1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

ECHILD	The calling process has no child processes. This condition implies that the calling process was not suspended by `wait3`.
EFAULT	The *status* or *rusage* arguments point to an illegal address. This condition implies that the calling process was not suspended by `wait3`.

SEE ALSO

`exec`(2), `_exit`(2), `fork`(2), `ptrace`(2), `sigpause`(2), `sigvec`(2), `sigaction`(2), `wait`(2), `wait3`(2), `exit`(3C).

STANDARDS

When using `m88kbcs` as the Software Development Environment target, the `wait4` function will be an incomplete emulation of Berkeley semantics. Since we are using BCS system calls, resource usage information is not available. If *rusage is non-NULL,* `wait4()` will fail with `errno` set to EINVAL.

NAME

`waitid` – wait for child process to change state

SYNOPSIS

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

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

where:

idtype `P_PID`, `P_GID`, or `P_ALL`
id A process identifier
infop A structure to contain information
options 0 or a positive integer (see "Option Flags" below)

DESCRIPTION

`Waitid` suspends the calling process until one of its children changes state. It records the current state of a child in the structure pointed to by *infop*. If a child process changed state prior to the call to `waitid`, `waitid` returns immediately.

The *idtype* and *id* arguments specify which children `waitid` is to wait for.

If *idtype* is `P_PID`, `waitid` waits for the child with a process ID equal to `(pid_t)` *id*.

If *idtype* is `P_PGID`, `waitid` waits for any child with a process group ID equal to `(pid_t)`*id*.

If *idtype* is `P_ALL`, `waitid` waits for any children and *id* is ignored.

Option Flags

The *options* argument is used to specify which state changes `waitid` is to wait for. It is formed by an OR of any of the following flags:

`WEXITED` Wait for process(es) to exit.

`WTRAPPED` Wait for traced process(es) to become trapped or reach a breakpoint [see `ptrace`(2) or dg_xtrace(2)].

`WSTOPPED` Wait for and return the process status of any child that has stopped upon receipt of a signal.

`WCONTINUED` Return the status for any child that was stopped and has been continued.

`WNOHANG` Return immediately.

`WNOWAIT` Keep the process in a waitable state.

infop must point to a `siginfo_t` structure, as defined in `siginfo`(5). `siginfo_t` is filled in by the system with the status of the process being waited for.

ACCESS CONTROL

No access checking is performed.

RETURN VALUE

If `waitid` returns due to a change of state of one of its children, a value of 0 is returned. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

DIAGNOSTICS

`EFAULT` *infop* points to an invalid address.

`EINTR` `waitid` was interrupted due to the receipt of a signal by the calling process.

EINVAL	An invalid value was specified for *options*.
EINVAL	*idtype* and *id* specify an invalid set of processes.
ECHILD	The set of processes specified by *idtype* and *id* does not contain any unwaited-for processes.

SEE ALSO

intro(2), exec(2), exit(2), fork(2), pause(2), ptrace(2), dg_xtrace(2), signal(2), sigaction(2), wait(2), siginfo(5).

NAME

write – write to an object

SYNOPSIS

```
int      write (fildes, buffer, nbyte)
int      fildes;
char     buffer[];
unsigned nbyte;
```

where:

fildes	An active, valid file descriptor.
buffer	User data buffer.
nbyte	Size (in bytes) of the user data buffer.

DESCRIPTION

`Write` transfers *nbyte* bytes of data from the buffer pointed to by *buffer* into the object associated with *fildes*.

If *fildes* refers to an object pointer having a current position attribute and the O_APPEND flag is clear, the write starts at a position in the object given by that attribute.

If *fildes* refers to an object pointer having a current position attribute and the O_APPEND flag is set, the position attribute of the object is set equal to the object's current size, where the write will start.

If the object pointer has no position attribute, then the starting write position depends on the type of object being written.

The behavior of the write call is affected by the object attribute flag O_NDELAY [see `open(2)`] associated with *fildes*.

The behavior of writes to a pipe or FIFO depends on whether or not the request is for more than PIPE_BUF bytes. Write requests of `PIPE_BUF` bytes or less are guaranteed not to be interleaved with data from other processes doing writes on the same pipe. Writes of greater than `PIPE_BUF` bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the `O_NONBLOCK` or `O_NDELAY` flags are set. Also, if a request is greater than `PIPE_BUF` bytes and all data previously written to the pipe has been read, `write` will transfer at least `PIPE_BUF` bytes.

If a write of *nbyte* bytes to a pipe (or FIFO) is requested, and *nbyte* is less than PIPE_BUF bytes, but *nbyte* of free space is currently not available in the pipe, then the following occurs:

> If the O_NDELAY and O_NONBLOCK flags are clear, the process will block until at least *nbyte* bytes of free space becomes available in the pipe, and the write will take place.
>
> If the O_NONBLOCK flag is set, –1 is returned and `errno` is set to EAGAIN. If both O_NONBLOCK and O_NDELAY are set, O_NONBLOCK has precedence.
>
> If the O_NDELAY flag is set, 0 is returned.

If a write of more than PIPE_BUF bytes is requested, the following occurs:

> If the O_NDELAY and O_NONBLOCK flags are clear, the process will block if the pipe is full. As space becomes available in the pipe, the data

from the write request will be written piecemeal–in multiple smaller amounts until the request is fulfilled. Thus, data from a write request of more than PIPE_BUF bytes may be interleaved on arbitrary byte boundaries with data written by other processes.

If the O_NONBLOCK flag is set and the pipe is full, the process will not block, −1 is returned with `errno` set to EAGAIN. If both O_NONBLOCK and O_NDELAY are set, O_NONBLOCK has precedence.

If the O_NONBLOCK flag is set and the pipe is not full, the process will not block, and as much data as will currently fit in the pipe will be written and that number of bytes is returned.

If the O_NDELAY is set and the pipe is full, the process will not block, and 0 is returned.

If the O_NDELAY flag is set and the pipe is not full, the process will not block, and as much data as will currently fit in the pipe will be written and that number of bytes is returned.

For STREAMS files [see `intro`(2)], the operation of `write` is determined by the values of the minimum and maximum *nbyte* range ("packet size") accepted by the stream. These values are contained in the topmost stream module. If *nbyte* falls within the packet size range, *nbyte* bytes are written. If *nbyte* does not fall within the range and the minimum packet size value is zero, `write` breaks the buffer into maximum packet size segments prior to sending the data downstream (the last segment may be smaller than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, `write` fails and sets `errno` to ERANGE. Writing a zero-length buffer (*nbyte* is zero) to a STREAMS device sends a zero length message with zero returned. However, writing a zero-length buffer to a pipe or FIFO sends no message and zero is returned. The user program may issue the `I_SWROPT` `ioctl`(2) to enable zero-length messages to be sent across the pipe or FIFO [see `streamio`(7)].

When writing to a stream, data messages are created with a priority band of zero.

The behavior of the write call is affected by the object attribute flag O_SYNC associated with *fildes*. This flag causes the `write` call to block until both the file data and file status are physically updated.

When `write` completes, the position attribute, if it exists, is incremented by the number of bytes actually written. The modification time for the file and the changed time for the file status are updated to reflect the time the write occurred.

If an error occurs, any changes to the object associated with *fildes* is defined by the object's type. The default situation is that the object associated with *fildes* is unchanged. This may not be the case for some errors on some types of objects.

If write is successful and O_SYNC is not specified, the data transferred may not be transferred to long term storage (in the case of an 'ordinary-disk-file' for example). To ensure this is the case, the `fsync` operation should be used.

ACCESS CONTROL

Fildes must be open for writing.

RETURN VALUE

0..*nbyte* Completed successfully. *nbyte* is the number of bytes actually written.

−1 An error occurred. errno is set to indicate the error.

DIAGNOSTICS

Errno may be set to one of the following error codes:

EBADF *Fildes* is not a valid file descriptor open for writing.

ERANGE if attempts to write to a stream with *nbyte* are outside the specified minimum and maximum write range, and the minimum value is non-zero. if the process is a member of a background process group and is 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.

EAGAIN The O_NDELAY flag was set and there was not enough room in the pipe.

EPIPE and SIGPIPE signal
An attempt is made to write to a pipe that is not open for reading or a socket of type SOCK_STREAM that is not connected to a peer socket.

EFBIG An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.

EFAULT *Buffer* points outside the process's allocated address space.

EINTR A signal was caught during the write system call.

ENOLCK A lock required to complete the call cannot be allocated from the system lock table.

EDEADLK *fildes* refers to a file that has mandatory record locking enabled and the read would produce a deadlock condition.

SEE ALSO

creat(2), dup(2), dup2(2), fcntl(2), ioctl(2), lseek(2), open(2), pipe(2), select(2), socket(2), socketpair(2), ulimit(2), writev(2).

NAME

writev – write on a file

SYNOPSIS

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

int    writev (fildes, iov, iovcnt)
int    fildes;
struct iovec iov[];
int    iovcnt;
```

where:

fildes	An active, valid file descriptor
iov	An array of extents
iovcnt	The number of extents given

DESCRIPTION

The `writev` system call transfers data from the *iovlen* buffers specified by members of the *iov* array: *iov*[0], *iov*[1], ..., *iov*[*iovlen*–1] into the object associated with *fildes*.

For `writev`, the *iovec* structure is defined as:

```
struct iovec {
  caddr_t  iov_base;
  int      iov_len;
  };
```

Each *iov* member specifies the base address and length of an area in memory where data is located. The `writev` call uses an area completely before proceeding to the next.

Iovcnt must be a positive number less than or equal to a system-imposed limit guaranteed to be at least MAXIOVCNT. The length of each extent (*iov_len*) in *iov*[] must be non-negative and the sum of these lengths must not overflow a 'long'.

Except for the disposition of the data, `writev` is equivalent to `write`.

ACCESS CONTROL

Fildes must be open for writing.

RETURN VALUE

0..*nbyte*	Completed successfully. The number of bytes actually written is returned. Here, *nbyte* is the sum of lengths of the *iovcnt* extents given in *iov*[].
–1	An error occurred. `errno` is set to indicate the error.

DIAGNOSTICS

`Errno` may be set to one of the following error codes:

EBADF *Fildes* is not a valid file descriptor open for writing.

EPIPE and SIGPIPE signal
: An attempt is made to write to a pipe not open for writing or a socket of type SOCK_STREAM that is not connected to a peer socket.

EFBIG	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.
EAGAIN	The O_NDELAY flag was set and there was not enough room in the pipe.
EDEADLK	*Fildes* refers to a file that has mandatory record locking enabled and the read would produce a deadlock condition.
ENOLCK	A lock required to complete the call cannot be allocated from the system lock table.
EINTR	A signal was caught during the system call.
EFAULT	*Iov* points outside the allocated address space.
EFAULT	One or more of the *iov* `[]` members point outside the allocated address space.
EINVAL	*Iovcnt* was invalid.
EINVAL	One or more of the *iov_len* values in *iov* `[]` was negative.
EINVAL	The sum of the *iov_len* values in *iov* `[]` overflowed a 'long'.

SEE ALSO

`creat(2)`, `dup(2)`, `dup2(2)`, `fcntl(2)`, `ioctl(2)`, `lseek(2)`, `open(2)`, `pipe(2)`, `select(2)`, `socket(2)`, `socketpair(2)`, `ulimit(2)`, `write(2)`.

End of Chapter

Index

Note: Boldfaced page numbers (e.g., **1-5**) indicate definitions of terms or other key information.

Related Documents

The following list of related manuals gives titles of Data General manuals followed by nine-digit numbers used for ordering. You can order any of these manuals via mail or telephone (see the TIPS Order Form in the back of this manual).

For a complete list of AViiON® and DG/UX™ manuals, see the *Guide to AViiON® and DG/UX™ Documentation* (069-701085). The on-line version of this manual found in **/usr/release/doc_guide** contains the most current list.

Data General Software Manuals

User's Manuals

User's Reference for the DG/UX™ System
Contains an alphabetical listing of manual pages for commands relating to general system operation. Ordering Number — 093-701054

Using the DG/UX™ Editors
Describes the text editors **vi** and **ed**, the batch editor **sed**, and the command line editor **editread**. Ordering Number — 069-701036

Using the DG/UX™ System
Describes the DG/UX system and its major features, including the C and Bourne shells, typical user commands, the file system, and communications facilities such as **mailx**. Ordering Number — 069-701035

Installation and Administration Manuals

System Manager's Reference for the DG/UX™ System
Contains an alphabetical listing of manual pages for commands relating to system administration or operation. Ordering Number — 093-701050

Programming Manuals

Porting and Developing Applications on the DG/UX™ System
A compendium of useful information for experienced programmers developing or porting applications to the DG/UX™ system. It includes information on how to: set up your environment, use the software development tools, compile and link programs, port to the windowing environment, and build BCS applications. It also describes available debuggers and the various industry standards the DG/UX system supports. Ordering Number — 069-701059

Programmer's Guide: ANSI C and Programming Support Tools (UNIX System V Release 4)
Describes the standard tools of the UNIX program development environment including compiling, linking, debugging, and analysis and revision control. An accompanying supplement, *Supplement for Programmer's Guide: ANSI C and Programming Support Tools* (086-000180) describes the DG/UX system enhancements and differences. Ordering Number — 093-701104

Programmer's Guide: Systems Services and Application Packaging Tools (UNIX System V Release 4)
Describes standard programming procedures and interfaces available to the C application developer in the UNIX environment. Topics include interprocess communications, memory management, file and record locking and application packaging. Note: Chapters 5 and 9 of this Prentice Hall manual discuss topics that do not apply to the DG/UX system. Ordering Number — 093-701103

Programmer's Reference for the DG/UX™ System, (Volume 2)
Alphabetical listing of manual pages for DG/UX subroutines and libraries. This is part of a three-volume set. Ordering Number — 093-701056

Programmer's Reference for the DG/UX™ System, (Volume 3)
Alphabetical listing of manual pages for DG/UX file formats, miscellaneous features, and networking protocols. Part of a three-volume set, this volume contains the table of contents and index (**contents** (0) and **index** (0)) for man pages. Ordering Number — 093-701102

Programming in the DG/UX™ Kernel Environment
Introduces kernel-level programming on the DG/UX™ system and provides reference pages for kernel-supplied utility routines. This manual is a pre-requisite to both *UNIX System V Release 4: Programmer's Guide: STREAMS* and *Writing a Standard Device Driver for the DG/UX™ System*. Ordering Number — 093-701083

End of Related Documents

TO ORDER

1. An order can be placed with the TIPS group in two ways:
   a) MAIL ORDER – Use the order form on the opposite page and fill in all requested information. Be sure to inclu shipping charges and local sales tax. If applicable, write in your tax exempt number in the space provided on the or form.

   Send your order form with payment to: Data General Corporation
   ATTN: Educational Services/TIPS G155
   4400 Computer Drive
   Westboro, MA 01581–9973

   b) TELEPHONE – Call TIPS at (508) 870–1600 for all orders that will be charged by credit card or paid for by purch orders over $50.00. Operators are available from 8:30 AM to 5:00 PM EST.

METHOD OF PAYMENT

2. As a customer, you have several payment options:
   a) Purchase Order – Minimum of $50. If ordering by mail, a hard copy of the purchase order must accompany ord
   b) Check or Money Order – Make payable to Data General Corporation.
   c) Credit Card – A minimum order of $20 is required for Mastercard or Visa orders.

SHIPPING

3. To determine the charge for UPS shipping and handling, check the total quantity of units in your order and refer to th following chart:

Total Quantity	Shipping & Handling Charge
1–4 Units	$5.00
5–10 Units	$8.00
11–40 Units	$10.00
41–200 Units	$30.00
Over 200 Units	$100.00

If overnight or second day shipment is desired, this information should be indicated on the order form. A separate char will be determined at time of shipment and added to your bill.

VOLUME DISCOUNTS

4. The TIPS discount schedule is based upon the total value of the order.

Order Amount	Discount
$1–$149.99	0%
$150–$499.99	10%
Over $500	20%

TERMS AND CONDITIONS

5. Read the TIPS terms and conditions on the reverse side of the order form carefully. These must be adhered to at all tim

DELIVERY

6. Allow at least two weeks for delivery.

RETURNS

7. Items ordered through the TIPS catalog may not be returned for credit.
8. Order discrepancies must be reported within 15 days of shipment date. Contact your TIPS Administrator at (508) 870–16 to notify the TIPS department of any problems.

INTERNATIONAL ORDERS

9. Customers outside of the United States must obtain documentation from their local Data General Subsidiary or Representative. Any TIPS orders received by Data General U.S. Headquarters will be forwarded to the appropriate D Subsidiary or Representative for processing.

TIPS ORDER FORM

Mail To: Data General Corporation
Attn: Educational Services/TIPS G155
4400 Computer Drive
Westboro, MA 01581 - 9973

BILL TO:	SHIP TO: (No P.O. Boxes - Complete Only If Different Address)
COMPANY NAME ____	COMPANY NAME ____
ATTN: ____	ATTN: ____
ADDRESS ____	ADDRESS (NO PO BOXES) ____
CITY ____	CITY ____
STATE ____ ZIP ____	STATE ____ ZIP ____

Priority Code ____________ (See label on back of catalog)

Authorized Signature of Buyer (Agrees to terms & conditions on reverse side) ____ Title ____ Date ____ Phone (Area Code) ____ E

ORDER #	QTY	DESCRIPTION	UNIT PRICE	TO PR

A

☐ UPS	ADD
1-4 Items	$ 5.00
5-10 Items	$ 8.00
11-40 Items	$ 10.00
41-200 Items	$ 30.00
200+ Items	$100.00

Check for faster delivery

Additional charge to be determined at time of shipment and added to your bill.

☐ UPS Blue Label (2 day shipping)

☐ Red Label (overnight shipping)

B VOLUME DISCOUNTS

Order Amount	Save
$0 – $149.99	0%
$150 – $499.99	10%
Over $500.00	20%

Tax Exempt # or Sales Tax (if applicable) ____

ORDER TOTAL	
Less Discount See B	–
SUB TOTAL	
Your local* sales tax	+
Shipping and handling – See A	+
TOTAL – See C	

C PAYMENT METHOD

☐ Purchase Order Attached ($50 minimum)
P.O. number is ________. (Include hardcopy P.O.)

☐ Check or Money Order Enclosed

☐ Visa ☐ MasterCard ($20 minimum on credit cards)

Account Number ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐ Expiration Date ☐☐☐☐

Authorized Signature ____________

(Credit card orders without signature and expiration date cannot be processed.)

THANK YOU FOR YOUR ORDER

PRICES SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.
PLEASE ALLOW 2 WEEKS FOR DELIVERY.
NO REFUNDS NO RETURNS.

* Data General is required by law to collect applicable sales or use tax on a purchases shipped to states where DG maintains a place of business, wh covers all 50 states. Please include your local taxes when determining th value of your order. If you are uncertain about the correct tax amount, ple call 508-870-1600.

Form 702
Rev. 8/87

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form. These terms and conditions apply to all orders, telephone, telex, or mail. By accepting these products the Customer accepts and agrees to be bound by these terms and conditions.

1. CUSTOMER CERTIFICATION
Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

2. TAXES
Customer shall be responsible for all taxes, including taxes paid or payable by DGC for products or services supplied under this Agreement, exclusive of taxes based on DGC's net income, unless Customer provides written proof of exemption.

3. DATA AND PROPRIETARY RIGHTS
Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

4. LIMITED MEDIA WARRANTY
DGC warrants the CLI Macros media, provided by DGC to the Customer under this Agreement, against physical defects for a period of ninety (90) days from the date of shipment by DGC. DGC will replace defective media at no charge to you, provided it is returned postage prepaid to DGC within the ninety (90) day warranty period. This shall be your exclusive remedy and DGC's sole obligation and liability for defective media. This limited media warranty does not apply if the media has been damaged by accident, abuse or misuse.

5. DISCLAIMER OF WARRANTY
EXCEPT FOR THE LIMITED MEDIA WARRANTY NOTED ABOVE, DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS, CLI MACROS OR MATERIALS SUPPLIED HEREUNDER.

6. LIMITATION OF LIABILITY
A. CUSTOMER AGREES THAT DGC'S LIABILITY, IF ANY, FOR DAMAGES, INCLUDING BUT NOT LIMITED TO LIABILITY ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT LIABILITY IN TORT OR WARRANTY SHALL NOT EXCEED THE CHARGES PAID BY CUSTOMER FOR THE PARTICULAR PUBLICATION OR CLI MACRO INVOLVED. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO CLAIMS FOR PERSONAL INJURY CAUSED SOLELY BY DGC'S NEGLIGENCE. OTHER THAN THE CHARGES REFERENCED HEREIN, IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOST PROFITS AND DAMAGES RESULTING FROM LOSS OF USE, OR LOST DATA, OR DELIVERY DELAYS, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY THEREOF; OR FOR ANY CLAIM BY ANY THIRD PARTY.

B. ANY ACTION AGAINST DGC MUST BE COMMENCED WITHIN ONE (1) YEAR AFTER THE CAUSE OF ACTION ACCRUES.

7. GENERAL
A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts, excluding its conflict of law rules. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer. DGC hereby rejects all such different, conflicting, or additional terms.

8. IMPORTANT NOTICE REGARDING AOS/VS INTERNALS SERIES (ORDER #1865 & #1875)
Customer understands that information and material presented in the AOS/VS Internals Series documents may be specific to a particular revision of the product. Consequently user programs or systems based on this information and material may be revision-locked and may not function properly with prior or future revisions of the product. Therefore, Data General makes no representations as to the utility of this information and material beyond the current revision level which is the subject of the manual. Any use thereof by you or your company is at your own risk. Data General disclaims any liability arising from any such use and I and my company (Customer) hold Data General completely harmless therefrom.

Cut here and insert in binder spine pocket

Data General

Data General Corporation, Westboro, Massachusetts 01580