1. INTRODUCTION

This manual describes the programming features of SYSTEM V/68. It provides neither a general overview of the operating system nor details of the implementation of the system. Not all of the features described are contained in the basic operating system; some are provided with unbundled utility packages. These may usually be identified by the page header; for example, "Networking Support Utilities" or "Security Administration Utilities."

This manual is divided into five sections, some containing interfiled subclasses:

1. Commands
2. System Calls
3. Subroutines
   - 3C. C Programming Language Libraries
   - 3S. Standard I/O Library Routines
   - 3M. Mathematical Library Routines
   - 3N. Networking Support Utilities
   - 3X. Specialized Libraries
4. File Formats
5. Miscellaneous Facilities.

Section 1 (*Commands*) describes commands that support C and other programming languages.

Section 2 (*System Calls*) describes the access to the services provided by the operating system kernel, including the C language interface.

Section 3 (*Subroutines*) describes the available subroutines. 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.

Section 4 (*File Formats*) 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**.

Section 5 (*Miscellaneous Facilities*) contains a variety of things. Included are descriptions of character sets, macro packages, etc.

References with numbers other than those above mean that the utility is contained in the appropriate section of another manual. References with **(1)** following the command mean that the utility is contained in this manual or the *User's Reference Manual*. Those followed by **(1M), (7),** or **(8)** are contained in the *System Administrator's Reference Manual*.

Each section consists of a number of independent entries of a page or so each. Entries within each section are alphabetized, with the exception of the introductory entry that begins each section (also Section 3 is in alphabetical order by suffixes). Some entries may describe several routines, commands, etc. In such cases, the entry appears only once, alphabetized under its "primary" name, the name that appears at the upper corners of each manual page.

All entries are based on a common format, not all of whose parts always appear:

- The **NAME** part gives the name(s) of the entry and briefly states its purpose.
- The **SYNOPSIS** part summarizes the use of the program being described. A few conventions are used, particularly in Section 2 (*System Calls*):
  - **Boldface** strings are literals and are to be typed just as they appear.
  - *Italic* strings usually represent substitutable argument prototypes and program names found elsewhere in the manual.
  - Square brackets **[ ]** around an argument prototype indicate that the argument is optional. When an argument prototype is given as "name" or "file," it always refers to a *file* name.
  - Ellipses **...** are used to show that the previous argument prototype may be repeated.
  - A final convention is used by the commands themselves. An argument beginning with a minus, plus, or equal sign (**-**, **+**, **=**) is often taken to be some sort of flag argument, even if it appears in a position where a file name could appear. Therefore, it is unwise to have filenames beginning with -, +, or =.
- The **DESCRIPTION** part describes the utility.

- The **EXAMPLES** part gives examples of usage, where appropriate.
- The **FILES** part gives the file names that are built into the program.
- The **SEE ALSO** part gives pointers to related information.
- The **DIAGNOSTICS** part discusses the diagnostic indications that may be produced. Messages that are intended to be self-explanatory are not listed.
- The **NOTES** part gives generally helpful hints about the use of the utility.
- The **WARNINGS** part points out potential pitfalls.
- The **BUGS** part gives known bugs and sometimes deficiencies.
- The **CAVEATS** part gives details of the implementation that might affect usage.

A Permuted Index is provided at the back of this manual. This is a list of keywords, given in the second of three columns, together with the context in which each keyword is found. The right column lists the name of the manual page on which each keyword may be found. The left column contains useful information about the keyword.

Some of the manual pages refer to the MC68881, which is a Motorola Floating Point Coprocessor Acceleration Unit. This unit is a high performance floating point support processor chip that supports a large subset of IEEE P754 Draft 10 requirements for Binary Floating Point Arithmetic.

NAME

intro – introduction to commands and application programs

DESCRIPTION

This section describes, in alphabetical order, commands available for the Motorola VME-based computers. Certain distinctions of purpose are made in the headings.

The following Utility packages are delivered with the computer:

Basic Networking Utilities
Cartridge Tape Controller Utilities
Directory and File Management Utilities
Editing Utilities
Essential Utilities
Help Utilities
Inter-process Communications
Line Printer Spooling Utilities
Performance Measurement Utilities
Spell Utilities
Terminal Filters Utilities
Terminal Information Utilities
User Environment Utilities

The following Utility Packages are (or will be) available for purchase:

Board Software Extension Package
Document Processing Package
International Support Package
Networking Support Utilities
Remote File Sharing Utilities
Security Administration Utilities

Manual Page Command Syntax

Unless otherwise noted, commands described in the **SYNOPSIS** section of a manual page accept options and other arguments according to the following syntax and should be interpreted as explained below.

name [*–option*...] [*cmdarg*...]
where:

[ ]	Surround an *option* or *cmdarg* that is not required.
...	Indicates multiple occurrences of the *option* or *cmdarg*.
name	The name of an executable file.
option	(Always preceded by a "–".) *noargletter*... or, *argletter optarg*[,...]
noargletter	A single letter representing an option without an option-argument. Note that more than one *noargletter* option can be grouped after one "–" (Rule 5, below).
argletter	A single letter representing an option requiring an option-argument.
optarg	An option-argument (character string) satisfying a preceding *argletter*. Note that groups of *optargs* following an *argletter* must be separated by commas, or separated by white space and quoted (Rule 8, below).
cmdarg	Path name (or other command argument) *not* beginning with "–", or "–" by itself indicating the standard input.

Command Syntax Standard: Rules

These command syntax rules are not followed by all current commands, but all new commands will obey them. *getopts*(1) should be used by all shell procedures to parse positional parameters and to check for legal options. It supports Rules 3-10 below. The enforcement of the other rules must be done by the command itself.

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.

SEE ALSO

getopts(1).
exit(2), wait(2), getopt(3C) in the *Programmer's Reference Manual.*
How to Get Started, at the front of this document.

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, or bad or inaccessible data. It is called variously "exit code", "exit status", or "return code", and is described only where special conventions are involved.

WARNINGS

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]] [–rrel] [–t[name]] [–fflag[flag-val]][–dflag[flag-val]] [–alogin] [–elogin] [–m[mrlist]] [–y[comment]] [–h] [–z] files

DESCRIPTION

admin is used to create new SCCS files and change parameters of existing ones. Arguments to *admin*, which may appear in any order, consist of keyletter arguments, which begin with –, and named files (note that SCCS file names must begin with the characters **s.**). If a named file does not exist, it is created, and its parameters are initialized according to the specified keyletter arguments. Parameters not initialized by a keyletter argument are assigned a default value. If a named file does exist, parameters corresponding to specified keyletter arguments are changed, and other parameters are left as is.

If a directory is named, *admin* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed since the effects of the arguments apply independently to each named file.

–n This keyletter indicates that a new SCCS file is to be created.

–i*[name]* The *name* of a file from which the text for a new SCCS file is to be taken. The text constitutes the first delta of the file (see **–r** keyletter for delta numbering scheme). If the **i** keyletter is used, but the file name is omitted, the text is obtained by reading the standard input until an end-of-file is encountered. If this keyletter is omitted, then the SCCS file is created empty. Only one SCCS file may be created by an *admin* command on which the **i** keyletter is supplied. Using a single *admin* to create

two or more SCCS files requires that they be created empty (no **–i** keyletter). Note that the **–i** keyletter implies the **–n** keyletter.

–r*rel* The *rel*ease into which the initial delta is inserted. This keyletter may be used only if the **–i** keyletter is also used. If the **–r** keyletter is not used, the initial delta is inserted into release 1. The level of the initial delta is always 1 (by default initial deltas are named 1.1).

–t*[name]* The *name* of a file from which descriptive text for the SCCS file is to be taken. If the **–t** keyletter is used and *admin* is creating a new SCCS file (the **–n** and/or **–i** keyletters also used), the descriptive text file name must also be supplied. In the case of existing SCCS files: (1) a **–t** keyletter without a file name causes removal of descriptive text (if any) currently in the SCCS file, and (2) a **–t** keyletter with a file name causes text (if any) in the named file to replace the descriptive text (if any) currently in the SCCS file.

–f*flag* This keyletter specifies a *flag*, and, possibly, a value for the *flag*, to be placed in the SCCS file. Several **f** keyletters may be supplied on a single *admin* command line. The allowable *flag*s and their values are:

b Allows use of the **–b** keyletter on a *get*(1) command to create branch deltas.

c*ceil* The highest release (i.e., "ceiling"), a number greater than 0 but less than or equal to 9999, which may be retrieved by a *get*(1) command for editing. The default value for an unspecified **c** flag is 9999.

f*floor* The lowest release (i.e., "floor"), a number greater than 0 but less than 9999, which may be retrieved by a *get*(1) command for editing. The default value for an unspecified **f** flag is 1.

d*SID* The default delta number (SIDs+1) to be used by a *get*(1) command.

i*[str]* Causes the "No id keywords (ge6)" message issued by *get*(1) or *delta*(1) to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords [see *get*(1)] are found in the text retrieved or stored in the SCCS file. If a value is supplied, the keywords must exactly match the given string, however the string must contain a keyword, and no embedded newlines.

j Allows concurrent *get*(1) commands for editing on the same SIDs+1 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>::= | **a**

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

n Causes *delta*(1) to create a "null" delta in each of those releases (if any) being skipped when a delta is made in a *new* release (e.g., in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as "anchor points" so that branch deltas may later be created from them. The absence of this flag causes skipped releases to be nonexistent in the SCCS file, preventing branch deltas from being created from them in the future.

q*text* User definable text substituted for all occurrences of the %Q% keyword in SCCS file text retrieved by *get*(1).

m*mod* *Mod*ule name of the SCCS file substituted for all occurrences of the %M% keyword in SCCS file text retrieved by *get*(1). If the **m** flag is not specified, the value assigned is the name of the SCCS file with the leading **s.** removed.

t*type* *Type* of module in the SCCS file substituted for all occurrences of %Y% keyword in SCCS file text retrieved by *get*(1).

v*pgm* Causes *delta*(1) to prompt for Modification Request (MR) numbers as the reason for creating a delta. The optional value specifies the name of an MR number validity checking program [see *delta*(1)]. (If this flag is set when creating an SCCS file, the **m** keyletter must also be used even if its value is null).

–d*flag* Causes removal (deletion) of the specified *flag* from an SCCS file. The **–d** keyletter may be specified only when processing existing SCCS files. Several **–d** keyletters may be supplied on a single *admin* command. See the **–f** keyletter for allowable *flag* names.

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

–a*login* A *login* name, or numerical system group ID, to be added to the list of users which may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all *login* names common to that group ID. Several **a** keyletters may be used on a single *admin* command line. As many *login*s, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas. If *login* or group ID is preceded by a **!** they are to be denied permission to make deltas.

–e*login* A *login* name, or numerical group ID, to be erased from the list of users allowed to make deltas (changes) to the SCCS file. Specifying a group ID is equivalent to specifying all *login* names common to that group ID. Several **e** keyletters may be used on a single *admin* command line.

–m*[mrlist]* The list of Modification Requests (MR) numbers is inserted into the SCCS file as the reason for creating the initial delta in a manner identical to *delta*(1). The **v** flag must be set and the MR numbers are validated if the **v** flag has a value (the name of an MR number validation program). Diagnostics will occur if the **v** flag is not set or MR validation fails.

–y*[comment]* The *comment* text is inserted into the SCCS file as a comment for the initial delta in a manner identical to that of *delta*(1). Omission of the **–y** keyletter results in a default comment line being inserted in the form:

date and time created *YY/MM/DD HH:MM:SS* by *login*

The **–y** keyletter is valid only if the **–i** and/or **–n** keyletters are specified (i.e., a new SCCS file is being created).

–h Causes *admin* to check the structure of the SCCS file [see *sccsfile*(5)], and to compare a newly computed check-sum (the sum of all the characters in the SCCS file except those in the first line) with the check-sum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced. keyletter inhibits writing on the file, so that it nullifies the effect of any other keyletters supplied, and is, therefore, only meaningful when processing existing files.

–z The SCCS file check-sum is recomputed and stored in the first line of the SCCS file (see **–h**, above).

Note that use of this keyletter on a truly corrupted file may prevent future detection of the corruption.

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

It is recommended that directories containing SCCS files be mode 755 and that SCCS files themselves be mode 444. The mode of the directories allows only the owner to modify SCCS files contained in the directories. The mode of the SCCS files prevents any modification at all except by SCCS commands.

If it should be necessary to patch an SCCS file for any reason, the mode may be changed to 644 by the owner allowing use of *ed*(1). *Care must be taken!* The edited file should *always* be processed by an **admin –h** to check for corruption followed by an **admin –z** to generate a proper check-sum. Another **admin –h** is recommended to ensure the SCCS file is valid.

admin also makes use of a transient lock file (called **z.***file-name*), which is used to prevent simultaneous updates to the SCCS file by different users. See *get*(1) for further information.

FILES

g-file	Existed before the execution of *delta*; removed after completion of *delta*.
p-file	Existed before the execution of *delta*; may exist after completion of *delta*.
q-file	Created during the execution of *delta*; removed after completion of *delta*.
x-file	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z-file	Created during the execution of *delta*; removed during the execution of *delta*.
d-file	Created during the execution of *delta*; removed after completion of *delta*.
/usr/bin/bdiff	Program to compute differences between the "gotten" file and the *g-file*.

SEE ALSO

delta(1), get(1), prs(1), what(1), sccsfile(4).
ed(1), help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

SEE ALSO

ld(1), lorder(1), strip(1), tmpnam(3S), a.out(4), ar(4) in the *Programmer's Reference Manual.*

NOTES

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

NAME

as, as00, as10, as20, as30 — common MC68xxx series processor assembler

SYNOPSIS

as [options] filename
as[name] [options] filename
where [name] may be 00, 10, 20, or 30

DESCRIPTION

The *as* command assembles the named file. The coprocessor MC68881 instructions are always legal. The coprocessor MC68851 instructions are always legal except when the MC68030 is selected, in which case only the MC68030 memory management instructions are recognized and the other MC68851 instructions are given warnings. For explanations of the name variations of the assember, see the **–p** option below.

The following flags may be specified in any order:

–o *objfile* Put the output of the assembly in *objfile*. By default, the output file name is formed by removing the **.s** suffix, if there is one, from the input file name and appending a **.o** suffix.

–p *name* Specify processor prejudice or exlusivity: *name* may be **000, 010, 020,** or **030** for the MC68000, MC68010, MC68020, or MC68030, respectively. Note: The command **as00 a.s** is equivalent to the command **as -p 000 a.s.**

More than one **–p** option will cause all but the last **–p** option to be ignored.

If the unadorned *as* command is used, no **-p** option is given. The assembler looks at the shell environment variable PROCESSOR for one of the values M68030, M68020, M68010 or M68000 and defaults to a prejudice toward that processor. However, if the PROCESSOR environment variable is missing or set to an unrecognized value, the default built into the assembler is to the native processor.

-n or **-j** Turn off long/short address optimization. By default, address optimization takes place.

–m Run the *m4* macro processor on the input to the assembler.

–R Remove (unlink) the input file after assembly is completed.

–dl Do not produce line number information in the object file.

–V Write the version number of the assembler being run on the standard error output.

–x Assemble instructions and address modes for the designated (or default) processor only. Give errors instead of warnings for instructions and addressing modes not available on the designated processor. This option does not apply to M68881 instructions but will give errors to some M68851 instructions and addressing modes when the M68030 is selected.

FILES

TMPDIR/* temporary files

TMPDIR is usually /usr/tmp but can be redefined by setting the environment variable **TMPDIR** [see *tempnam*() in *tmpnam*(3S)].

The assembler will create as many as sixteen (16) temporary files in **TMPDIR** which will be named according to a convention most easily described by the regular expression

as[1-9A-F][A-O]AAa[0-3][0-9]{4}

some examples of which follow:

as1AAAa03232
asAAAAa29455
asFOAAa01011

The last five digits are the process number of the *as* command which created the temporary file.

SEE ALSO

cc(1), ld(1), m4(1), nm(1), strip(1), tmpnam(3S), a.out(4)
Chapter 18, *SYSTEM V/68 Programmer's Guide*, MU43815PG/D2.

WARNINGS

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 are assembler symbols and which are real *m4* macros.

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

BUGS

The **.align** assembler directive may not work in the **.text** section when optimization is performed.

NOTES

Wherever possible, the assembler should be accessed through a compilation system interface program [such as *cc*(1)].

NAME

cb – C program beautifier

SYNOPSIS

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

DESCRIPTION

The *cb* comand reads C programs either from its arguments or from the standard input, and writes them on the standard output with spacing and indentation that display the structure of the code. Under default options, *cb* preserves all user new-lines.

cb accepts the following options.

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

–j Causes split lines to be put back together.

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

SEE ALSO

cc(1).

Kernighan, B. W., and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, 1978.

BUGS

Punctuation that is hidden in preprocessor statements will cause indentation errors.

NAME

cc – C compiler

SYNOPSIS

cc [options] files

DESCRIPTION

The *cc* command is the interface to the C Compilation System. The compilation tools consist of a preprocessor, compiler, optimizer, assembler and link editor. The *cc* command processes the supplied options and then executes the various tools with the proper arguments. The *cc* command accepts several types of files as arguments:

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 runs through the assembler then an object program is produced and is left in the file whose name is that of the source with **.o** substituted for **.c**. However, the **.o** file is normally deleted if a single C program is compiled and then immediately link edited. In the same way, files whose names end in **.s** are taken to be assembly source programs, and may be assembled and link edited; and files whose names end 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.

Since the *cc* command usually creates files in the current directory during the compilation process, it is necessary to run the *cc* command in a directory in which a file can be created.

The following options are interpreted by *cc*: .

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

–g Cause the compiler to generate additional information needed for the use of *sdb* (1).

–o *outfile*
Produce an output object file by the name *outfile*. The name of the default file is **a.out**. This is a link editor option.

–p Arrange for the compiler to produce code that counts the number of times each routine is called; also, if link editing takes place, profiled versions of libc.a and libm.a (with –lm option) are linked and *monitor (3C)* is automatically called. A **mon.out** file will then be produced at normal termination of execution of the object program. An execution profile can then be generated by use of *prof (1)*.

–B*string*

–t*[p02al]*

These options will be removed in the next release. Use the **–Y** option.

–E Run only *cpp (1)* on the named C programs, and send the result to the standard output.

–H Print out on *stderr* the pathname of each file included during the current compilation.

–O Invoke an intermediate-code and/or object-code optimizer. The environment variable **OPTIM** controls which of these optimizers will be invoked. If **OPTIM=HL** (High Level), then only the intermediate-code optimizer will be invoked. If **OPTIM=BOTH**, then both optimizers will be invoked. If **OPTIM** is not set to either of these values, then only the object-code optimizer will be invoked. This flag cannot be used with the **–g** flag.

–P Run only *cpp (1)* on the named C programs and and leave the result in corresponding files suffixed **.i**. This option is passed to *cpp (1)*.

–S Compile and do not assemble the named C programs, and leave the assembler-language output in corresponding files suffixed **.s**.

–V Print the version of the compiler, optimizer, assembler and/or link editor that is invoked.

–Wc,arg1[,arg2...]

Hand off the argument[s] *argi* to pass *c* where *c* is one of [**p0o12al**] interpreted as for -Y, below. For example: **–Wa,–m** passes **–m** to the assembler.

–Y [p0o12alSILU],*dirname*

Specify a new pathname, *dirname*, for the locations of the tools and directories designated in the first argument. **[po1alSILU]** represents:

p preprocessor
0 compiler - first pass
o optional intermediate code (high-level) optimizer
1 compiler - second pass
2 object-code (peephole) optimizer
a assembler
l link editor
S directory containing the start-up routines
I default include directory searched by *cpp (1)*
L first default library directory searched by *ld (1)*
U second default library directory searched by *ld (1)*

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

The *cc* command also recognizes –C, –D, –H, –I and –U and passes these options and their arguments directly to the preprocessor without using the –W option. Similarly, the *cc* command recognizes –a, –l, –m, –o, –r, –s, –t, –u, –x, –z, –L, –M and –V and passes these options and their arguments directly to the loader. See the manual pages for *cpp (1)* and *ld (1)* for descriptions.

In addition, *cc* provides the ability to align most variables on double word (four-byte) boundaries. This is done to improve performance by eliminating double memory accesses required to obtain data that straddles a double word boundary. Setting environment variable **DBLALIGN** to **YES** and exporting it provides this feature. This is the recommended setting for **DBLALIGN**. The user may override this alignment by setting **DBLALIGN** to **NO** and exporting it. Programs compiled with **DBLALIGN** set to **NO** will have their non-character variables aligned on word (two-byte) boundaries, which is what *cc*(1) does on MC68000 and MC68010 systems.

The environment variable **STALIGN** is closely allied to the environment variable **DBLALIGN**. Setting **DBLALIGN=YES** aligns most variables to double word (multiple of 4 byte) boundaries. However, variables within

structures are still only single-aligned to provide compatibility with existing file header formats and I/O data structures. Persons who want long-alignment within structures can set environment variable **STALIGN=YES** along with **DBLALIGN=YES**, but the C compiler generated code in this case will not be compatible with existing file headers or with standard libraries, such as **libc.a**. Obviously, the **STALIGN=YES** setting is of very limited use.

Users who have the MC68881 floating point chip may also set environment variable **FP** to **M68881** and export it. This will enable MC68881 code generation in the compiler and also automatically select appropriate MC68881 runtime routines at link time. In addition, the two floating point libraries must be linked with the object code; refer to the **–l** option of *ld*(1).

Other arguments are taken to be either link editor option arguments or C-compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C-compatible routines. These programs, together with the results of any compilations specified, are link-edited (in the order given) to produce an executable program with the name **a.out** unless the **–o** option of the link editor is used.

Other arguments are taken to be C compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C compatible routines and are passed directly to the link editor. These programs, together with the results of any compilations specified, are link edited (in the order given) to produce an executable program with name **a.out** unless the **–o** option of the link editor is used.

The user should be aware that floating point object code built with **FP=M68881** is incompatible with prior floating point object code. This is because the 68881 code returns "float" and "double" function return values in MC68881 register **%fp0**, rather than in **%d0** and **%d1** as non-68881 object did.

This incompatibility can be remedied at the expense of greater code size and longer execution time by setting environment variable **FP=M68881U**. This setting allows function return values to be returned in both floating point register **%fp0** and in M68000 family registers **%d0/%d1** during C compiler code generation. This allows new 68881-specific object to be linked with old (non-68881) C compiler floating point code. CAUTION: This setting is available only on the C compiler. The *f77* compiler does not support **FP=M68881U**.

Several versions of **libc.a** and **libm.a** are provided depending on the type of object code required. For systems without an MC68881 floating point unit, the libraries **libc.a** and **libm.a** have been built without 68881 code generation enabled. For users compiling C programs with **FP=M68881**, the compatible libraries **libc881.a** and **libm881.a** are available. When compiling C programs with **FP=M68881U**, the libraries **libc881u.a** and **libm881u.a** should be used.

**

If the cc command is put in a file *prefix*cc the prefix will be parsed off the command and used to call the tools, i.e., *prefix*tool. For example, OLDcc will call OLDcpp, OLDcomp, OLDoptim, OLDas, and OLDld and will link OLDcrt1.o. Therefore, one MUST be careful when moving the cc command around. The prefix will apply to the preprocessor, compiler, optimizer, assembler, link editor, and the start-up routines.

The C language standard was extended to allow arbitrary length variable names. The option pair "**–Wp,–T –W0,–XT**" will cause cc to truncate arbitrary length variable names.

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
LIBDIR/*crt0.o	start-up routine
LIBDIR/mcrt0.o	profiling start-up routine
TMPDIR/*	temporary files
LIBDIR/cpp	preprocessor, *cpp*(1)
LIBDIR/c[01]	compiler
LIBDIR/optim	optional peephole optimizer
LIBDIR/coptim	optional high-level optimizer
BINDIR/as	assembler, *as*(1)
BINDIR/ld	link editor, *ld*(1)
LIBDIR/libc.a	standard C library

TMPDIR/*	temporary files
LIBDIR/cpp	preprocessor, *cpp*(1)
LIBDIR/c[01]	compiler
LIBDIR/optim	optional peephole optimizer
LIBDIR/coptim	optional high-level optimizer
BINDIR/as	assembler, *as*(1)
BINDIR/ld	link editor, *ld*(1)
LIBDIR/libc.a	standard C library
LIBDIR/libc_s.a	standard C shared library
LIBDIR/libm.a	standard math library
LIBDIR/libc881.a	floating point library
LIBDIR/libc881u.a	floating point library
LIBDIR/libm881.a	floating point math library
LIBDIR/libm881u.a	floating point math library
LIBDIR/libp/lib*.a	profiled versions of libraries

LIBDIR is usually /lib
BINDIR is usually /bin
TMPDIR is usually /usr/tmp but can be redefined by setting the environment variable **TMPDIR** [see *tempnam*() in *tmpnam*(3S)].

SEE ALSO

as(1), ld(1), cpp(1), gencc(1), lint(1), prof(1), sdb(1), tmpnam(3S).
Kernighan, B. W., and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, 1978.

DIAGNOSTICS

The diagnostics produced by the C compiler are sometimes cryptic. Occasional messages may be produced by the assembler or link editor.

NOTES

By default, the return value from a compiled C program is completely random. The only two guaranteed ways to return a specific value is to explicitly call *exit*(2) or to leave the function **main**() with a *"return expression;"* construct.

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 (SCCS IDentification string) specified by the **–r** keyletter, of each named SCCS file.

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

If a directory is named, *cdc* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read (see *WARNINGS*) and each line of the standard input is taken to be the name of an SCCS file to be processed.

Arguments to *cdc*, which may appear in any order, consist of *keyletter* arguments and file names.

All the described *keyletter* arguments apply independently to each named file:

–r*SID* Used to specify the *S*CCS *ID*entification (SID) string of a delta for which the delta commentary is to be changed.

–m*mrlist* If the SCCS file has the **v** flag set [see *admin*(1)] then a list of MR numbers to be added and/or deleted in the delta commentary of the SID specified by the **–r** keyletter *may* be supplied. A null MR list has no effect.

MR entries are added to the list of MRs in the same manner as that of *delta*(1). In order to delete an MR, precede the MR number with the character **!** (see *EXAMPLES*). If the MR to be deleted is currently in the list of MRs, it is removed and changed into a "comment" line. A list of all deleted MRs is placed in the comment section of the delta commentary and preceded by a comment line stating that they were deleted.

If **–m** is not used and the standard input is a terminal, the prompt **MRs?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The **MRs?** prompt always precedes the **comments?** prompt (see **–y** keyletter).

MRs in a list are separated by blanks and/or tab characters. An unescaped new-line character terminates the MR list.

Note that if the **v** flag has a value [see *admin*(1)], it is taken to be the name of a program (or shell procedure) which validates the correctness of the MR numbers. If a non-zero exit status is returned from the MR number validation program, *cdc* terminates and the delta commentary remains unchanged.

–y*[comment]* Arbitrary text used to replace the *comment*(s) already existing for the delta specified by the **–r** keyletter. The previous comments are kept and preceded by a comment line stating that they were changed. A null *comment* has no effect.

If **–y** is not specified and the standard input is a terminal, the prompt **comments?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped new-line character terminates the *comment* text.

Simply stated, the keyletter arguments are either (1) if you made the delta, you can change its delta commentary; or (2) if you own the file and

directory you can modify the delta commentary.

EXAMPLES

cdc –r1.6 –m"bl78-12345 !bl77-54321 bl79-00001" –ytrouble s.file

adds bl78-12345 and bl79-00001 to the MR list, removes bl77-54321 from the MR list, and adds the comment **trouble** to delta 1.6 of s.file.

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

does the same thing.

WARNINGS

If SCCS file names are supplied to the *cdc* command via the standard input (– on the command line), then the **–m** and **–y** keyletters must also be used.

FILES

x-file [see *delta*(1)]
z-file [see *delta*(1)]

SEE ALSO

admin(1), delta(1), get(1), prs(1), sccsfile(4).
help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

cflow – generate C flowgraph

SYNOPSIS

cflow [–r] [–ix] [–i_] [–dnum] files

DESCRIPTION

The *cflow* command analyzes a collection of C, yacc, lex, assembler, and object files and attempts to build a graph charting the external references. Files suffixed with **.y**, **.l**, and **.c** are yacced, lexed, and C-preprocessed as appropriate. The results of the preprocessed files, and files suffixed with **.i**, are then run through the first pass of *lint*(1). Files suffixed with **.s** are assembled. Assembled files, and files suffixed with **.o**, have information extracted from their symbol tables. The results are collected and turned into a graph of external references which is displayed upon the standard output.

Each line of output begins with a reference number, followed by a suitable number of tabs indicating the level, then the name of the global symbol followed by a colon and its definition. Normally only function names that do not begin with an underscore are listed (see the **–i** options below). For information extracted from C source, the definition consists of an abstract type declaration (e.g., **char** *), and, delimited by angle brackets, the name of the source file and the line number where the definition was found. Definitions extracted from object files indicate the file name and location counter under which the symbol appeared (e.g., *text*). Leading underscores in C-style external names are deleted.

Once a definition of a name has been printed, subsequent references to that name contain only the reference number of the line where the definition may be found. For undefined references, only < > is printed.

As an example, given the following in *file.c*:

```
int     i;

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

```
f()
{
        i = h();
}
```

the command

```
cflow -ix file.c
```

produces the output

```
1       main: int(), <file.c 4>
2               f: int(), <file.c 11>
3                       h: <>
4                       i: int, <file.c 1>
5               g: <>
```

When the nesting level becomes too deep, the output of *cflow* can be piped to *pr*(1), using the **-e** option, to compress the tab expansion to something less than every eight spaces.

In addition to the **-D**, **-I**, and **-U** options [which are interpreted just as they are by *cc*(1) and *cpp*(1)], the following options are interpreted by *cflow*:

-r Reverse the "caller:callee" relationship producing an inverted listing showing the callers of each function. The listing is also sorted in lexicographical order by callee.

-ix Include external and static data symbols. The default is to include only functions in the flowgraph.

-i_ Include names that begin with an underscore. The default is to exclude these functions (and data if *-ix* is used).

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

DIAGNOSTICS

Complains about bad options. Complains about multiple definitions and only believes the first. Other messages may come from the various programs used (e.g., the C-preprocessor).

SEE ALSO

as(1), cc(1), cpp(1), lex(1), lint(1), nm(1), yacc(1).
pr(1) in the *User's Reference Manual.*

BUGS

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

NAME

comb – combine SCCS deltas

SYNOPSIS

comb files

DESCRIPTION

comb generates a shell procedure [see *sh*(1)] which, when run, will reconstruct the given SCCS files. The reconstructed files will, hopefully, be smaller than the original files. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *comb* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored. The generated shell procedure is written on the standard output.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed, but the effects of any keyletter argument apply independently to each named file. each **get –e** generated, this argument causes the reconstructed file to be accessed at the release of the delta to be created, otherwise the reconstructed file would be accessed at the most recent ancestor. Use of the **–o** keyletter may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file. argument causes *comb* to generate a shell procedure which, when run, will produce a report giving, for each file: the file name, size (in blocks) after combining, original size (also in blocks), and percentage change computed by:

100 * (original – combined) / original

It is recommended that before any SCCS files are actually combined, one should use this option to determine exactly how much space is saved by the combining process. *S*CCS *ID*entification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file. *list* (see *get*(1) for the syntax of a *list*) of deltas to be preserved. All other deltas are discarded.

If no keyletter arguments are specified, *comb* will preserve only leaf deltas and the minimal number of ancestors needed to preserve the tree.

FILES

s.COMB	The name of the reconstructed SCCS file.
comb?????	Temporary.

SEE ALSO

admin(1), delta(1), get(1), prs(1), sccsfile(4).
help(1), sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

comb may rearrange the shape of the tree of deltas. It may not save any space; in fact, it is possible for the reconstructed file to actually be larger than the original.

NAME

cpp – the C language preprocessor

SYNOPSIS

LIBDIR/cpp [option ...] [ifile [ofile]]

DESCRIPTION

The C language preprocessor, *cpp*, is invoked as the first pass of any C compilation by the *cc*(1) command. Thus *cpp*'s output is designed to be in a form acceptable as input to the next pass of the C compiler. As the C language evolves, *cpp* and the rest of the C compilation package will be modified to follow these changes. Therefore, the use of *cpp* other than through the *cc*(1) command is not suggested, since the functionality of *cpp* may someday be moved elsewhere. See *m4*(1) for a general macro processor.

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.

The following *options* to *cpp* are recognized:

–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*, where *name* is a reserved symbol that is predefined by the particular preprocessor. Following is the current list of these possibly reserved symbols. In SYSTEM V/68, *sysV68*, *unix* and *m68k* are defined.

operating system:	unix, sysV68, dmert, gcos, ibm, os, tss
hardware:	interdata, pdp11, u370, u3b, u3b5, u3b2, u3b20d, vax, m68k
system variant:	RES, RT
lint(1):	lint

–D*name*
–D*name*=*def*
Define *name* with value *def* as if by a **#define**. If no =*def* is given, *name* is defined with value 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.

–T The **–T** option forces *cpp* to use only the first eight characters to distinguish preprocessor symbols and is included for backward compatibility.

–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.

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

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

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*. They can be used anywhere (including in macros) just as any other defined name.

All *cpp* directive lines start with # in column 1. Any number of blanks and tabs is 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*
Notice that there can be no space between *name* and the (. Replace subsequent instances of *name* followed by a (, a list of comma-separated sets of tokens, and a) followed by *token-string*, where each occurrence of an *arg* in the *token-string* is replaced by the corresponding set of tokens in the comma-separated 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* re-starts its scan for names to expand at the beginning of the newly created *token-string*.

#undef *name*

Cause the definition of *name* (if any) to be forgotten from now on. No additional tokens are permitted on the directive line after *name*.

#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 the *<filename>* notation is used, *filename* is only searched for in the standard places. See the **–I** and **–Y** options above for more detail. No additional tokens are permitted on the directive line after the final " or >.

#line *integer-constant "filename"*

Causes *cpp* to generate line control information for the next pass of the C compiler. *Integer-constant* is the line number of the next line and *filename* is the file from which it comes. If *"filename"* is not given, the current file name is unchanged. No additional tokens are permitted on the directive line after the optional *filename*.

#endif

Ends a section of lines begun by a test directive (**#if**, **#ifdef**, or **#ifndef**). Each test directive must have a matching **#endif**. No additional tokens are permitted on the directive line.

#ifdef *name*

The lines following will appear in the output if and only if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**. No additional tokens are permitted on the directive line after *name*.

#ifndef *name*

The lines following will appear in the output if and only if *name* has not been the subject of a previous **#define**. No additional tokens are permitted on the directive line after *name*.

#if *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 by the C language. There is also a unary operator **defined**, which can be used in *constant-expression* in these two forms: **defined** (*name*) or **defined** *name*. This allows the utility of **#ifdef** and **#ifndef** in a **#if** directive. Only these operators, integer constants, and names which are known by *cpp* should be used in *constant-expression*. In particular, the **sizeof** operator is not available.

To test whether either of two symbols, *foo* and *fum*, are defined, use

```
#if defined(foo) | defined(fum)
```

#elif *constant-expression*

An arbitrary number of **#elif** directives is allowed between a **#if**, **#ifdef**, or **#ifndef** directive and a **#else** or **#endif** directive. The lines following the **#elif** directive will appear in the output if and only if the preceding test directive evaluates to zero, all intervening **#elif** directives evaluate to zero, and the *constant-expression* evaluates to non-zero. If *constant-expression* evaluates to non-zero, all succeeding **#elif** and **#else** directives will be ignored. Any *constant-expression* allowed in a **#if** directive is allowed in a **#elif** directive.

#else The lines following will appear in the output if and only if the preceding test directive evaluates to zero, and all intervening **#elif** directives evaluate to zero. No additional tokens are permitted on the directive line.

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

FILES

INCDIR	standard directory list for **#include** files, usually /usr/include
LIBDIR	usually /lib

SEE ALSO

cc(1), lint(1), m4(1).

DIAGNOSTICS

The error messages produced by *cpp* are intended to be self-explanatory. The line number and file name where the error occurred are printed along with the diagnostic.

NOTES

The unsupported **–W** option enables the **#class** directive. If it encounters a **#class** directive, *cpp* will exit with code 27 after finishing all other processing. This option provides support for "C with classes".

Because the standard directory for included files may be different in different environments, this form of **#include** directive:

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

should be used, rather than one with an absolute path, like:

```
#include "/usr/include/file.h"
```

cpp warns about the use of the absolute pathname.

NAME

cprs – compress a common object file

SYNOPSIS

cprs [**–p**] file1 file2

DESCRIPTION

The *cprs* command reduces the size of a common object file, *file1*, by removing duplicate structure and union descriptors. The reduced file, *file2*, is produced as output.

The sole option to *cprs* is:

–p Print statistical messages including: total number of tags, total duplicate tags, and total reduction of *file1*.

SEE ALSO

strip(1), a.out(4), syms(4).

NAME

create - create master release media utility, R3.1

DESCRIPTION

The *create* program creates a set of master distribution media for application software products. The new version, R3.1, is similar to previous versions but is more robust and tolerant of operator errors.

One new feature of *create* is the automatic creation of a file containing the cyclic redundancy checksum (crc), length, and modification time for each file included on the distribution media. A copy of this file is put on the media and transferred to a customer's system, while another version is derived from *upgrade*(1M) for cross-checking purposes.

MEDIA CREATED WITH THIS RELEASE OF CREATE CAN ONLY BE READ BY R3.1 AND LATER VERSIONS OF UPGRADE!

EXAMPLE

This section describes the interactive dialogue between a user and *create*. Indented text represents the program display screens; indented **boldface** text represents sample user input; *italicized* text represents variable items. Text shown in braces (for example, {Yes, No}) represents a choice of responses to the prompts. Any response that is too long is truncated; you are then prompted to verify what was entered. Text shown in square brackets ([ ]) is commentary that never appears on the screen.

To execute the *create* program, type:

$ create

The screen is now cleared via 'tput clear'.

Create Master Release Media Utility, R3.1
Copyright 1984,86 by Motorola, Inc.

What type of media is this product distributed on?
{Floppy disk, 1.2MB floppy, Tape cartridge, Cartridge disk, 9-track tape}
-->**xxx**

Choose the appropriate device name from the list shown in braces; type the device name (or the first character of the name), followed by

RETURN. Pressing only RETURN causes the program to select the first item in the list. An invalid input, such as **xxx** shown here, produces the following error message:

```
You must enter the first letter of one of the choices and a RETURN,
or just RETURN to select the first choice.
```

The previous message is then redisplayed.

```
What type of media is this product distributed on?
{Floppy disk, 1.2MB floppy, Tape cartridge, Cartridge disk, 9-track tape}
-->c    [for a cartridge disk]
```

If the media needs to be formatted before it can be used, the following questions are asked:

```
If you like, I will automatically format the media for you.
Would you like this?
{Yes, No}
--> y

Enter the shell command needed to format this type of media:
-->
```

The particular command needed here will vary, depending upon the device type, the system in use, its configuration, etc. The following virtual device names are normally linked to their respective raw device names in */dev*, and should be allowable in the response:

/dev/FLOPPY	for 640K floppy disk
/dev/FLOPPY.MB	for 1.2MB floppy disk
/dev/IOMEGA	for 5MB Iomega cartridge disks

If specified, this command is issued to the shell each time a new volume of media is mounted.

```
Mount media volume #1, then hit RETURN...
```

If automatic formatting was requested, the specified format command is displayed now (not shown here) as it is invoked.

What is the name of this product? --> **example**

What is this product's new release identifier? --> **EXAMPLE3.0**

Enter a file name (<11 chars; similar to the product name) for auditing need
--> **example (#2)**

This name is used for auditing purposes and must conform to file system naming conventions. It is a good idea to use some variant of the product's name here with no embedded spaces or special characters. If illegal characters are entered, the following warning message is displayed and the prompt is re-issued:

This name must conform to SYSTEM V/68 file naming conventions!!!

Enter a file name (<11 chars; similar to the product name) for auditing need
--> **example**

Enter the pathname for the directory where this product is found
(you're now in <*name-of-current-working-directory*>)
--> **xyz**

Any absolute or valid relative pathname is legal here. If you are already in the root directory for the product, just enter a period (.). If an invalid directory name is entered, such as the xyz shown, the following message is displayed:

create: warning -- 'xyz' is an invalid directory path!!!
(errno = 2)

Enter the pathname for the directory where this product is found
(you're now in /usr/local/src/example/common)
--> . .
(The product's root pathname is /usr/local/src/example)

Into what directory (on the TARGET SYSTEM) should Upgrade copy this pro
-->

This is where the product will be installed in a user's system when *upgrade* is run. A null response is not allowed, and causes the prompt to be repeated. There are two other types of invalid responses which cause the following error messages to appear on the screen:

create: warning -- The target directory name must begin with a '/'!

create: warning -- The target directory name cannot be just '/'!

Into what directory (on the TARGET SYSTEM) should Upgrade copy this product?
--> **/d31/work/example** [A valid response]

Note that if this directory path doesn't exist in the USER'S SYSTEM, *upgrade* will create it, if possible.

Enter a command string (<128 chars) to be executed at START of upgrade:
--> **date**

This command, if specified, is submitted to the shell by *upgrade* before any of the product's files are read from the distribution media. In this example, the *date* command is given although any sequence of commands and/or scripts is legal.

Enter a command string (<128 chars) to be executed at END of upgrade:
-->

Similarly, this optional command string is executed by *upgrade* after all of the product's files have been copied into the specified TARGET SYSTEM directory. For either command string, a null response is accepted, as in this example.

Please stand-by for a moment while I figure some things out...

(nnnn media blocks needed) . . . [updated periodically]
(deriving an audit file now) . . . [displayed when *create* generates a crc file]

The audit files are placed into the */usr/AUDITS* directory for use by *upgrade*. Specifically, the *.cr0 file is copied to the distribution media for later installation into a user's system for auditing needs.

When this phase is finished, the screen is cleared via 'tput clear'.

```
    Create Master Release Media Utility, R3.1

This product's name is 'example'
It's release identifier is: EXAMPLE 3.0
(Auditing facilities will use the name: example)
It is located in the directory rooted at '/usr/local/src/example'.
It occupies about 292 x 1024-byte disk blocks in the file system,
    and requires 268 (1024-byte) blocks of distribution media.

The distribution media was specified as Cartridge disk.
So, 1 volume(s) of this media will be needed to distribute this product.

Upgrade will execute this command string prior to installation:
    date
It will then install the product into the directory named:
    /d3/work/example
Finally, this command string will be invoked after installation:
    --- none specified ---

Are these input parameters correct?
{Yes, No}
-->
```

A no (**n**) response will allow the user to re-define everything. A yes (y) or null response will start the creation of the distribution media.

Note that the byte count shown in the fourth line above (i.e., "292 x 1024-byte blocks") will vary depending on whether the file system containing the product's files uses 512 or 1024 byte blocks. Both *create* and *upgrade* select the appropriate number for the system. The distribution media is always written with 1024 byte blocks.

```
        ------ FILE COPY IN PROGRESS ------

nnnn blocks written to vol #1 from /usr/local/src/example.
nnnn blocks being verified on media . . .
```

The two lines above show *create*'s progress as it copies the product's files

to the media and re-reads the media to verify its integrity.

If more than one media volume is needed, the following prompt is issued after each volume has been filled:

Mount media volume #n, then hit RETURN...

Automatic formatting is performed now, if specified (not shown here).

nnnn blocks written to volume #n from <source_directory>...
nnnn blocks being verified on media . . .

and so on, until the entire product has been written to media.

Now creating an audit file for this product...

Finished creating master(s) for 'example (EXAMPLE 3.0)'.

(A table-of-contents listing of this product with crc's is contained in:
'<audit_file_name>'

Do you wish to create a master copy of another product?
{No, Yes}
-->
If you're done, you can just hit RETURN now.

Pressing RETURN terminates program execution.

DIAGNOSTICS

Most parameters are validated before being committed for use, either by the program or by the user via the status display. It is possible, however, for an internally executed shell command to die, in which case an error message may or may not be produced. Known error conditions fall into two categories: warnings and fatal errors. Warnings produce a message, but allow for continued execution. Fatal errors include any error that would prevent a complete creation of the distribution media. Warning and fatal error messages are shown below.

Note that if the */usr/bin/crc* program is missing, *create* should be aborted and re-run after *crc*(1M) is located and installed.

WARNINGS

Common warnings:

The target directory name must begin with a '/'
The target directory cannot be just a '/'
The media failed verification
'<pathname>' is an invalid directory path

Less common warnings:

can't determine size of audit file
can't read/verify end-of-volume flag from media
can't read/verify checksum block from media

FATAL ERRORS

Common fatal error:

You must be logged in as root or have root's setuid permission set!

Less common fatal errors:

can't open device {<dev_name>} for writing
error encountered while writing header info
error writing to output media
error reading from input data stream
error encountered while writing end-of-volume flag
error encountered while writing checksum to the media
unable to open output media for reading
error encountered while reading header
error occurred attempting to read data from media
The end-of-volume flag block is corrupted
cannot chdir to '<source_root_directory>'

FILES

tput
/usr/bin/crc
/usr/AUDITS/*

SEE ALSO

create(4).
upgrade(1M), crc(1M) in the *System Administrator's Reference Manual.*
tput(1) in the *User's Reference Manual.*

NAME

ctrace – C program debugger

SYNOPSIS

ctrace [options] [file]

DESCRIPTION

The *ctrace* command allows you to follow the execution of a C program, statement-by-statement. The effect is similar to executing a shell procedure with the **–x** option. *ctrace* reads the C program in *file* (or from standard input if you do not specify *file*), inserts statements to print the text of each executable statement and the values of all variables referenced or modified, and writes the modified program to the standard output. You must put the output of *ctrace* into a temporary file because the *cc*(1) command does not allow the use of a pipe. You then compile and execute this file.

As each statement in the program executes it will be listed at the terminal, followed by the name and value of any variables referenced or modified in the statement, followed by any output from the statement. Loops in the trace output are detected and tracing is stopped until the loop is exited or a different sequence of statements within the loop is executed. A warning message is printed every 1000 times through the loop to help you detect infinite loops. The trace output goes to the standard output so you can put it into a file for examination with an editor or the *bfs*(1) or *tail*(1) commands.

The options commonly used are:

–f *functions* Trace only these *functions*.
–v *functions* Trace all but these *functions*.

You may want to add to the default formats for printing variables. Long and pointer variables are always printed as signed integers. Pointers to character arrays are also printed as strings if appropriate. Char, short, and int variables are also printed as signed integers and, if appropriate, as characters. Double variables are printed as floating point numbers in scientific notation. You can request that variables be printed in additional formats, if appropriate, with these options:

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

These options are used only in special circumstances:

–l *n* Check *n* consecutively executed statements for looping trace output, instead of the default of 20. Use 0 to get all the trace output from loops.

–s Suppress redundant trace output from simple assignment statements and string copy function calls. This option can hide a bug caused by use of the = operator in place of the == operator.

–t *n* Trace *n* variables per statement instead of the default of 10 (the maximum number is 20). The Diagnostics section explains when to use this option.

–P Run the C preprocessor on the input before tracing it. You can also use the **–D**, **–I**, and **–U** *cpp*(1) options.

These options are used to tailor the run-time trace package when the traced program will run in a non-SYSTEM V/68 System environment:

–b Use only basic functions in the trace code, that is, those in *ctype*(3C), *printf*(3S), and *string*(3C). These are usually available even in cross-compilers for microprocessors. In particular, this option is needed when the traced program runs under an operating system that does not have *signal*(2), *fflush*(3S), *longjmp*(3C), or *setjmp*(3C).

–p *string*
Change the trace print function from the default of 'printf('. For example, 'fprintf(stderr,' would send the trace to the standard error output.

–r *f* Use file *f* in place of the *runtime.c* trace function package. This lets you change the entire print function, instead of just the name and leading arguments (see the **–p** option).

EXAMPLE

If the file *lc.c* contains this C program:

```
1 #include <stdio.h>
2 main()        /* count lines in input */
3 {
4       int c, nl;
```

```
5
6       nl = 0;
7       while ((c = getchar()) != EOF)
8               if (c = '\n')
9                       ++nl;
10      printf("%d\n", nl);
11 }
```

and you enter these commands and test data:

```
cc lc.c
a.out
1
(cntl-d)
```

the program will be compiled and executed. The output of the program will be the number **2**, which is not correct because there is only one line in the test data. The error in this program is common, but subtle. If you invoke *ctrace* with these commands:

```
ctrace lc.c >temp.c
cc temp.c
a.out
```

the output will be:

```
 2 main()
 6      nl = 0;
        /* nl == 0 */
 7      while ((c = getchar()) != EOF)
```

The program is now waiting for input. If you enter the same test data as before, the output will be:

```
        /* c == 49 or '1' */
 8              if (c = '\n')
                /* c == 10 or '\n' */
 9                      ++nl;
                        /* nl == 1 */
 7      while ((c = getchar()) != EOF)
        /* c == 10 or '\n' */
 8              if (c = '\n')
                /* c == 10 or '\n' */
 9                      ++nl;
                        /* nl == 2 */
 7      while ((c = getchar()) != EOF)
```

If you now enter an end of file character (cntl-d) the final output will be:

```
        /* c == -1 */
```

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

Note that the program output printed at the end of the trace line for the **nl** variable. Also note the **return** comment added by *ctrace* at the end of the trace output. This shows the implicit return at the terminating brace in the function.

The trace output shows that variable **c** is assigned the value '1' in line 7, but in line 8 it has the value '\n'. Once your attention is drawn to this **if** statement, you will probably realize that you used the assignment operator (=) in place of the equality operator (==). You can easily miss this error during code reading.

EXECUTION-TIME TRACE CONTROL

The default operation for *ctrace* is to trace the entire program file, unless you use the **–f** or **–v** options to trace specific functions. This does not give you statement-by-statement control of the tracing, nor does it let you turn the tracing off and on when executing the traced program.

You can do both of these by adding *ctroff*() and *ctron*() function calls to your program to turn the tracing off and on, respectively, at execution time. Thus, you can code arbitrarily complex criteria for trace control with *if* statements, and you can even conditionally include this code because *ctrace* defines the **CTRACE** preprocessor variable. For example:

```
#ifdef CTRACE
        if (c == '!' && i > 1000)
                ctron();
#endif
```

You can also call these functions from *sdb*(1) if you compile with the **–g** option. For example, to trace all but lines 7 to 10 in the main function, enter:

```
sdb a.out
main:7b ctroff()
main:11b ctron()
r
```

You can also turn the trace off and on by setting static variable tr_ct_ to 0 and 1, respectively. This is useful if you are using a debugger that cannot

call these functions directly.

DIAGNOSTICS

This section contains diagnostic messages from both *ctrace* and *cc*(1), since the traced code often gets some *cc* warning messages. You can get *cc* error messages in some rare cases, all of which can be avoided.

ctrace Diagnostics

warning: some variables are not traced in this statement

Only 10 variables are traced in a statement to prevent the C compiler "out of tree space; simplify expression" error. Use the **–t** option to increase this number.

warning: statement too long to trace

This statement is over 400 characters long. Make sure that you are using tabs to indent your code, not spaces.

cannot handle preprocessor code, use –P option

This is usually caused by #ifdef/#endif preprocessor statements in the middle of a C statement, or by a semicolon at the end of a #define preprocessor statement.

'if ... else if' sequence too long

Split the sequence by removing an **else** from the middle.

possible syntax error, try –P option

Use the **–P** option to preprocess the *ctrace* input, along with any appropriate **–D**, **–I**, and **–U** preprocessor options. If you still get the error message, check the Warnings section below.

Cc Diagnostics

warning: illegal combination of pointer and integer
warning: statement not reached
warning: sizeof returns 0

Ignore these messages.

compiler takes size of function

See the *ctrace* "possible syntax error" message above.

yacc stack overflow

See the *ctrace* "'if ... else if' sequence too long" message above.

out of tree space; simplify expression

Use the **–t** option to reduce the number of traced variables per statement from the default of 10. Ignore the "ctrace: too many variables to trace" warnings you will now get.

redeclaration of signal
Either correct this declaration of *signal*(2), or remove it and #include <signal.h>.

SEE ALSO

signal(2), ctype(3C), fclose(3S), printf(3S), setjmp(3C), string(3C).
bfs(1), tail(1) in the *User's Reference Manual.*

WARNINGS

You will get a *ctrace* syntax error if you omit the semicolon at the end of the last element declaration in a structure or union, just before the right brace (}). This is optional in some C compilers.

Defining a function with the same name as a system function may cause a syntax error if the number of arguments is changed. Just use a different name.

ctrace assumes that BADMAG is a preprocessor macro, and that EOF and NULL are #defined constants. Declaring any of these to be variables, e.g., "int EOF;", will cause a syntax error.

BUGS

ctrace does not know about the components of aggregates like structures, unions, and arrays. It cannot choose a format to print all the components of an aggregate when an assignment is made to the entire aggregate. *ctrace* may choose to print the address of an aggregate or use the wrong format (e.g., 3.149050e-311 for a structure with two integer members) when printing the value of an aggregate.

Pointer values are always treated as pointers to character strings.

The loop trace output elimination is done separately for each file of a multi-file program. This can result in functions called from a loop still being traced, or the elimination of trace output from one function in a file until another in the same file is called.

FILES

/usr/lib/ctrace/runtime.c run-time trace package

NAME

cxref – generate C program cross-reference

SYNOPSIS

cxref [options] files

DESCRIPTION

The *cxref* command analyzes a collection of C files and attempts to build a cross-reference table. *cxref* uses a special version of *cpp* to include **#define**'d information in its symbol table. It produces a listing on standard output of all symbols (auto, static, and global) in each file separately, or, with the **–c** option, in combination. Each symbol contains an asterisk (*) before the declaring reference.

In addition to the **–D**, **–I** and **–U** options [which are interpreted just as they are by *cc*(1) and *cpp*(1)], the following *options* are interpreted by *cxref*:

–c Print a combined cross-reference of all input files.

–w<*num*>
Width option which formats output no wider than <num> (decimal) columns. This option will default to 80 if <num> is not specified or is less than 51.

–o file Direct output to *file*.

–s Operate silently; do not print input file names.

–t Format listing for 80-column width.

FILES

LLIBDIR	usually /usr/lib
LLIBDIR/xcpp	special version of the C preprocessor.

SEE ALSO

cc(1), cpp(1).

DIAGNOSTICS

Error messages are unusually cryptic, but usually mean that you cannot compile these files.

BUGS

cxref considers a formal argument in a *#define* macro definition to be a declaration of that symbol. For example, a program that *#includes* **ctype.h**, will contain many declarations of the variable **c**.

(1)

NAME

delta – make a delta (change) to an SCCS file

SYNOPSIS

delta [–rSID] [–s] [–n] [–glist] [–m[mrlist]] [–y[comment]] [–p] files

DESCRIPTION

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

delta makes a delta to each named SCCS file. If a directory is named, *delta* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read (see *WARNINGS*); each line of the standard input is taken to be the name of an SCCS file to be processed.

delta may issue prompts on the standard output depending upon certain keyletters specified and flags [see *admin*(1)] that may be present in the SCCS file (see **–m** and **–y** keyletters below).

Keyletter arguments apply independently to each named file.

–r*SID* Uniquely identifies which delta is to be made to the SCCS file. The use of this keyletter is necessary only if two or more outstanding *get*s for editing (**get –e**) on the same SCCS file were done by the same person (login name). The SID value specified with the **–r** keyletter can be either the SID specified on the *get* command line or the SID to be made as reported by the *get* command [see *get*(1)]. A diagnostic results if the specified SID is ambiguous, or, if necessary and omitted on the command line.

–s Suppresses the issue, on the standard output, of the created delta's SID, as well as the number of lines inserted, deleted and unchanged in the SCCS file.

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

–g*list* a *list* (see *get*(1) for the definition of *list*) of deltas which are to be *ignored* when the file is accessed at the change level (SID) created by this delta.

–m*[mrlist]* If the SCCS file has the **v** flag set [see *admin*(1)] then a Modification Request (MR) number *must* be supplied as the reason for creating the new delta.

If **–m** is not used and the standard input is a terminal, the prompt **MRs?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The **MRs?** prompt always precedes the **comments?** prompt (see **–y** keyletter).

MRs in a list are separated by blanks and/or tab characters. An unescaped new-line character terminates the MR list.

Note that if the **v** flag has a value [see *admin*(1)], it is taken to be the name of a program (or shell procedure) which will validate the correctness of the MR numbers. If a non-zero exit status is returned from the MR number validation program, *delta* terminates. (It is assumed that the MR numbers were not all valid.)

–y*[comment]* Arbitrary text used to describe the reason for making the delta. A null string is considered a valid *comment*.

If **–y** is not specified and the standard input is a terminal, the prompt **comments?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped new-line character terminates the comment text.

–p Causes *delta* to print (on the standard output) the SCCS file differences before and after the delta is applied in a *diff*(1) format.

FILES

g-file	Existed before the execution of *delta*; removed after completion of *delta*.
p-file	Existed before the execution of *delta*; may exist after completion of *delta*.
q-file	Created during the execution of *delta*; removed after completion of *delta*.
x-file	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z-file	Created during the execution of *delta*; removed during the execution of *delta*.
d-file	Created during the execution of *delta*; removed after completion of *delta*.
/usr/bin/bdiff	Program to compute differences between the "gotten" file and the *g-file*.

WARNINGS

Lines beginning with an **SOH** ASCII character (binary 001) cannot be placed in the SCCS file unless the **SOH** is escaped. This character has special meaning to SCCS [see *sccsfile*(4) (5)] and will cause an error.

A *get* of many SCCS files, followed by a *delta* of those files, should be avoided when the *get* generates a large amount of data. Instead, multiple *get*/*delta* sequences should be used.

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

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

SEE ALSO

admin(1), cdc(1), get(1), prs(1), rmdel(1), sccsfile(4).
bdiff(1), help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

dis – object code disassembler

SYNOPSIS

dis [**–o**] [**–V**] [**–L**] [**–d** sec] [**–da** sec] [**–F** function] [**–t** sec] [**–l** string] file ...

DESCRIPTION

The *dis* command produces an assembly language listing of *file*, which may be an object file or an archive of object files. The listing includes assembly statements and an octal or hexadecimal representation of the binary that produced those statements.

The following *options* are interpreted by the disassembler and may be specified in any order.

–o Print numbers in octal. The default is hexadecimal.

–V Print, on standard error, the version number of the disassembler being executed.

–L Lookup source labels in the symbol table for subsequent printing. This option works only if the file was compiled with additional debugging information [e.g., the **–g** option of *cc*(1)].

–d *sec* Disassemble the named section as data, printing the offset of the data from the beginning of the section.

–da *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.

–t *sec* Disassemble the named section as text.

–l *string* Disassemble the library file specified by *string*. For example, one would issue the command **dis –l x –l z** to disassemble **libx.a** and **libz.a**. All libraries are assumed to be in *LIBDIR*.

If the **–d**, **–da** 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]**, represents 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*(1)]. 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 **()**.

FILES

LIBDIR usually /lib.

SEE ALSO

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

DIAGNOSTICS

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

NAME

dump – dump selected parts of an object file

SYNOPSIS

dump [options] files

DESCRIPTION

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

This command will accept both object files and archives of object files. It processes each file argument according to one or more of the following options:

–a Dump the archive header of each member of each archive file argument.

–g Dump the global symbols in the symbol table of an archive.

–f Dump each file header.

–o Dump each optional header.

–h Dump section headers.

–s Dump section contents.

–r Dump relocation information.

–l Dump line number information.

–t Dump symbol table entries. Do not use this option on purely executable object code.

–z name Dump line number entries for the named function.

–c Dump the string table.

–L Interpret and print the contents of the *.lib* sections.

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

–d number Dump the section number, *number*, or the range of sections starting at *number* and ending at the *number* specified by **+d**.

+d number
Dump sections in the range either beginning with first section or beginning with section specified by **–d**.

–n name Dump information pertaining only to the named entity. This *modifier* applies to **–h**, **–s**, **–r**, **–l**, and **–t**.

–p Suppress printing of the headers.

–t index Dump only the indexed symbol table entry. The **–t** used in conjunction with **+t**, specifies a range of symbol table entries.

+t index Dump the symbol table entries in the range ending with the indexed entry. The range begins at the first symbol table entry or at the entry specified by the **–t** option.

–u Underline the name of the file for emphasis.

–v Dump information in symbolic representation rather than numeric (e.g., C_STATIC instead of 0X02). This *modifier* can be used with all the above options except **–s** and **–o** options of *dump*.

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

+z number Dump line numbers starting at either function *name* or *number* specified by **–z**, up to *number* specified by **+z**.

Blanks separating an *option* and its *modifier* are optional. The comma separating the name from the number modifying the **–z** option may be replaced by a blank.

The *dump* command attempts to format the information it dumps in a meaningful way, printing certain information in character, hex, octal or decimal representation as appropriate.

SEE ALSO

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

(1)

NAME

gcc – Green Hills C-68000 compiler

SYNOPSIS

gcc [option] ... file ...

DESCRIPTION

gcc is an optimizing SYSTEM V/68-compatible C compiler, optionally available for a Motorola VME-based computer system. *gcc* accepts several types of arguments:

Arguments whose names end with **.c** are C source programs; they are compiled and left in a **.o** file in the working directory.

Arguments whose names end with **.s** are assembly source programs; they are assembled and left in a **.o** file in the working directory.

Other arguments are taken to be loader option arguments, C-compatible object programs, or object program libraries.

The **.o** file is deleted if a single source file is compiled and linked.

The **.s** files which are created for each module are normally deleted.

gcc accepts many options; those listed below are of particular interest. Additional options are supported by *ld(1)*.

C "options"

–c Compile to the '.o' level only; do not load.

–v Verbose mode. The compiler driver will echo each command and options before exection.

–w Suppress warning messages.

–p Generate profiling code and link the code with routines which support *prof(1)*.

–pg Generate profiling code similar to **-p**, but link with a more involved profiling mechanism which supports *gprof*(1). This option forces the generation of frame pointers.

-g Generate symbolic debugger information in the assembly file, for use with a debugger such as *dbx(1)* or *cdb(1)*.

-ga Generate a frame for every routine, regardless of need. This often generates more compact code for medium and large routines.

-O Perform various speed optimizations, such as moving constant expressions out of loops. Generally this will make your programs somewhat larger; if their performance is not loop bound, they may become slower as well.

-OLM Allow the optimizer to assume that memory locations do not change except by explicit stores. That is, the optimizer is guaranteed that no memory locations are I/O device registers that can be changed by external hardware and no memory locations are being shared with other processes which can change them asynchronously with respect to the current process. This compile time option must be used with extreme caution (or not at all) in device drivers, operating systems, shared memory environments, and when interrupts (or SYSTEM V/68 signals) are present.

-R Make initialized variables part of the text segment; passed on to *as*.

-E Do not compile the program, instead place the output of the preprocessor on the standard output file. This is useful for debugging preprocessor macros. The integrated preprocessor cannot generate output as fast as *cpp*(1) , so use *cpp(1)* for big jobs.

-C Do not strip comments from the preprocessor output.

-S Compile the named programs, and leave the assembler-language output on corresponding files suffixed **.s**.

-o *output*
Name the final output file *output*. If this option is used the file **a.out** will be left undisturbed. This does not apply to assembly output.

-D*name*=*def*
-D*name* Define the *name* to the preprocessor, as if by '#define'. If no definition is given, the name is defined as "1".

–U*name* Remove any initial definition of *name*.

–I*dir* '#include' files whose names do not begin with / are always sought first in the directory of the *file* argument, then in directories named in –I options, then in */usr/include*.

–Xn Where n is an integer constant. Turn on option number n. There are numerous options available for such things as signed bit fields, short return types, etc. Many of these by nature should be performed on all programs which are linked together, and should therefore be built in to the compiler driver. Descriptions of these options can be found in the *User's Manual*. The following list may be used for quick reference.

–Zn Turn off option number n. This is the reverse of the X option. This option is useful for turning off options which are on by default.

–B*string* Use the compiler in the directory *string*, or use the default old version if *string* is blank.

–X14 Prepend underscores to all identifiers.

–X18 Only declared *register* variables are allocated in registers.

–X20 With **–X35** , no leading underscores.

–X21 With **–X35** , uppercase only.

–Z22 VAX format arithmetic.

–X23 Clear the stack after each routine call.

–X25 Display procedure sizes as compilation proceeds.

–X29 Output file should have a **.asm** suffix.

–X35 Generate Motorola format output, for cross development.

–X42 No single precision arithmetic calls.

–X92 SYSTEM V/68 style output.

-X96 Use space instead of blkb.

-X97 No .ascii statements in output.

-Z98 Disable 68020 code generation.

-Z99 Disable 68881 code generation.

-X120 Hunter and Ready position independence.

-X122 Generate binary for 68020/68881 instructions.

-X125 Activate **-X18** if there is a call to *setjmp*.

-Z129 Disable 68881 transcendental support. Calls to functions will not be mapped to instructions.

-Z130 Generate 68881 instructions in a manner compatible with old compilers.

-Z136 Disable Integrated Solutions assembler modifications.

-X138 Do not use *.lcomm* , use default space generation.

-X168 Do not move invariant floating point expressions out of loops.

-X211 Call the **strcpy()** library function.

-X243 Do not generate fsgldiv or fsglmul instructions.

FILES

file.c	C input file
file.o	object file
file.s	assembly file
a.out	loaded output
/usr/bin/gcc	
/usr/lib/ccom68	compiler
/usr/lib/crt0.o	runtime startoff
/usr/lib/mcrt0.o	startoff for profiling
/usr/lib/gcrt0.o	startoff for graph profiling
/lib/020/libc.a	standard C library
/usr/include	standard directory for '#include' files
mon.out	profile output for use by *prof*(1)
gmon.out	profile output for use by *gprof*(1)

SEE ALSO

B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, 1978

B. W. Kernighan, *Programming in C, Manual tutorial*

D. M. Ritchie, *C Reference Manual*
C-68000 User's Manual
as(1), prof(1), gprof(1), adb(1), cdb(1), ld(1)

DIAGNOSTICS

The diagnostics produced by the C compiler are intended to be self-explanatory, and similar to those produced by the SYSTEM V/68 C compiler. Occasional messages may be produced by the assembler or loader.

BUGS

Routines with over a thousand lines of code may require large amounts of memory to compile. If memory constraints are a problem do not use "-O" and break large routines into several smaller routines. Certain features may not be supported; see the System Release Guide.

NAME

gencc – create a front-end to the cc command

SYNOPSIS

gencc

DESCRIPTION

The *gencc* command is an interactive command designed to aid in the creation of a front-end to the *cc* command. Since hard-coded pathnames have been eliminated from the C Compilation System (CCS), it is possible to move pieces of the CCS to new locations without recompiling the CCS. The new locations of moved pieces can be specified through the **–Y** option to the *cc* command. However, it is inconvenient to supply the proper **–Y** options with every invocation of the *cc* command. Further, if a system administrator moves pieces of the CCS, such movement should be invisible to users.

The front-end to the *cc* command which *gencc* generates is a one-line shell script which calls the *cc* command with the proper **–Y** options specified. The front-end to the *cc* command will also pass all user supplied options to the *cc* command.

gencc prompts for the location of each tool and directory which can be respecified by a **–Y** option to the *cc* command. If no location is specified, it assumes that that piece of the CCS has not been relocated. After all the locations have been prompted for, *gencc* will create the front-end to the *cc* command.

gencc creates the front-end to the *cc* command in the current working directory and gives the file the same name as the *cc* command. Thus, *gencc* can not be run in the same directory containing the actual *cc* command. Further, if a system administrator has redistributed the CCS, the actual *cc* command should be placed somewhere which is not typically in a user's PATH (e.g., /lib). This will prevent users from accidentally invoking the *cc* command without using the front-end.

WARNINGS

gencc does not produce any warnings if a tool or directory does not exist at the specified location. Also, *gencc* does not actually move any files to new locations.

FILES

./cc	front-end to cc

SEE ALSO

cc(1).

NAME

get – get a version of an SCCS file

SYNOPSIS

get [–rSID] [–ccutoff] [–ilist] [–xlist] [–wstring] [–aseq-no.] [–**k**] [–**e**] [–**l**[p] [–**p**] [–**m**] [–**n**] [–**s**] [–**b**] [–**g**] [–**t**] file ...

DESCRIPTION

get generates an ASCII text file from each named SCCS file according to the specifications given by its keyletter arguments, which begin with –. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *get* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The generated text is normally written into a file called the *g-file* whose name is derived from the SCCS file name by simply removing the leading **s.**; (see also *FILES*, below).

Each of the keyletter arguments is explained below as though only one SCCS file is to be processed, but the effects of any keyletter argument applies independently to each named file.

–r*SID* The *S*CCS *ID*entification string (SID) of the version (delta) of an SCCS file to be retrieved. Table 1 below shows, for the most useful cases, what version of an SCCS file is retrieved (as well as the SID of the version to be eventually created by *delta*(1) if the **–e** keyletter is also used), as a function of the SID specified.

–c*cutoff* *Cutoff* date-time, in the form:

YY[MM[DD[HH[MM[SS]]]]]

No changes (deltas) to the SCCS file which were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, **–c7502** is equivalent to **–c750228235959**. Any number of non-numeric characters may separate the various 2-digit pieces of the *cutoff* date-time. This feature allows one to specify a

cutoff date in the form: "**–c77/2/2 9:22:25**". Note that this implies that one may use the %E% and %U% identification keywords (see below) for nested *gets* within, say the input to a *send*(1C) command:

```
~!get "–c%E% %U%" s.file
```

–i*list* A *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

```
<list> ::= <range> | <list> , <range>
<range> ::= SID | SID – SID
```

SID, the SCCS Identification of a delta, may be in any form shown in the "SID Specified" column of Table 1.

–x*list* A *list* of deltas to be excluded in the creation of the generated file. See the **–i** keyletter for the *list* format.

–e Indicates that the *get* is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of *delta*(1). The **–e** keyletter used in a *get* for a particular version (SID) of the SCCS file prevents further *get*s for editing on the same SID until *delta* is executed or the **j** (joint edit) flag is set in the SCCS file [see *admin*(1)]. Concurrent use of **get –e** for different SIDs is always allowed.

If the *g-file* generated by *get* with an **–e** keyletter is accidentally ruined in the process of editing it, it may be regenerated by re-executing the *get* command with the **–k** keyletter in place of the **–e** keyletter.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file [see *admin*(1)] are enforced when the **–e** keyletter is used.

–b Used with the **–e** keyletter to indicate that the new delta should have an SID in a new branch as shown in Table 1. This keyletter is ignored if the **b** flag is not present in the file [see *admin*(1)] or if the retrieved *delta* is not a leaf *delta*. (A leaf *delta* is one that has no successors on the SCCS file tree.)
Note: A branch *delta* may always be created from a non-leaf *delta*. Partial SIDs are interpreted as shown in the "SID Retrieved" column of Table 1.

–k Suppresses replacement of identification keywords (see below) in the retrieved text by their value. The **–k** keyletter is implied by the **–e** keyletter.

–l[p] Causes a delta summary to be written into an *l-file*. If **–lp** is used then an *l-file* is not created; the delta summary is written on the standard output instead. See *FILES* for the format of the *l-file*.

–p Causes the text retrieved from the SCCS file to be written on the standard output. No *g-file* is created. All output which normally goes to the standard output goes to file descriptor 2 instead, unless the **–s** keyletter is used, in which case it disappears.

–s Suppresses all output normally written on the standard output. However, fatal error messages (which always go to file descriptor 2) remain unaffected.

–m Causes each text line retrieved from the SCCS file to be preceded by the SID of the delta that inserted the text line in the SCCS file. The format is: SID, followed by a horizontal tab, followed by the text line.

–n Causes each generated text line to be preceded with the %M% identification keyword value (see below). The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the **–m** and **–n** keyletters are used, the format is: %M% value, followed by a horizontal tab, followed by the **–m** keyletter generated format.

–g Suppresses the actual retrieval of text from the SCCS file. It is primarily used to generate an *l-file*, or to verify the existence of a particular SID.

–t Used to access the most recently created delta in a given release (e.g., **–r1**), or release and level (e.g., **–r1.2**).

–w *string* Substitute *string* for all occurrences of %W% when getting the file.

–a*seq-no*. The delta sequence number of the SCCS file delta (version) to be retrieved [see *sccsfile*(5)]. This keyletter is used by the *comb*(1) command; it is not a generally useful keyletter. If both the **–r** and **–a** keyletters are specified, only the **–a** keyletter is used. Care should be taken when using the **–a** keyletter in conjunction with the **–e** keyletter, as the SID of the delta to be created may not be what one expects. The **–r** keyletter can be used with the **–a** and **–e** keyletters to control the naming of the SID of the delta to be created.

For each file processed, *get* responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the **–e** keyletter is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each file name is printed (preceded by a new-line) before it is processed. If the **–i** keyletter is used included deltas are listed following the notation "Included"; if the **–x** keyletter is used, excluded deltas are listed following the notation "Excluded".

TABLE 1. Determination of SCCS Identification String

SID* Specified	-b Keyletter Used†	Other Conditions	SID Retrieved	SID of Delta to be Created
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
R	no	R > mR	mR.mL	R.1***
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	–	R < mR and R does *not* exist	hR.mL**	hR.mL.(mB+1).1
R	–	Trunk succ.# in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L	no	No trunk succ.	R.L	R.(L+1)
R.L	yes	No trunk succ.	R.L	R.L.(mB+1).1
R.L	–	Trunk succ. in release ≥ R	R.L	R.L.(mB+1).1
R.L.B	no	No branch succ.	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch succ.	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch succ.	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch succ.	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch succ.	R.L.B.S	R.L.(mB+1).1

* "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB+1).1" means "the first sequence number on the *new* branch (i.e., maximum branch number plus one) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components *must* exist.

** "hR" is the highest *existing* release that is lower than the specified, *nonexistent*, release R.

*** This is used to force creation of the *first* delta in a *new* release.

Successor.

† The **–b** keyletter is effective only if the **b** flag [see *admin*(1)] is present in the file. An entry of – means "irrelevant".

‡ This case applies if the **d** (default SID) flag is *not* present in the file. If the **d** flag *is* present in the file, then the SID obtained from the **d** flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

IDENTIFICATION KEYWORDS

Identifying information is inserted into the text retrieved from the SCCS file by replacing *identification keywords* with their value wherever they occur. The following keywords may be used in the text stored in an SCCS file:

Keyword	*Value*
%M%	Module name: either the value of the **m** flag in the file [see *admin*(1)], or if absent, the name of the SCCS file with the leading **s.** removed.
%I%	SCCS identification (SID) (%R%.%L%.%B%.%S%) of the retrieved text.
%R%	Release.
%L%	Level.
%B%	Branch.
%S%	Sequence.
%D%	Current date (YY/MM/DD).
%H%	Current date (MM/DD/YY).
%T%	Current time (HH:MM:SS).
%E%	Date newest applied delta was created (YY/MM/DD).
%G%	Date newest applied delta was created (MM/DD/YY).
%U%	Time newest applied delta was created (HH:MM:SS).
%Y%	Module type: value of the **t** flag in the SCCS file [see *admin*(1)].
%F%	SCCS file name.
%P%	Fully qualified SCCS file name.
%Q%	The value of the **q** flag in the file [see *admin*(1)].

%C% Current line number. This keyword is intended for identifying messages output by the program such as "this should not have happened" type errors. It is *not* intended to be used on every line to provide sequence numbers.

%Z% The 4-character string **@(#)** recognizable by *what*(1).

%W% A shorthand notation for constructing *what*(1) strings for SYSTEM V/68 program files. %W% = %Z%%M%<horizontal-tab>%I%

%A% Another shorthand notation for constructing *what*(1) strings for non-SYSTEM V/68 program files.
%A% = %Z%%Y% %M% %I%%Z%

Several auxiliary files may be created by *get*. These files are known generically as the *g-file*, *l-file*, *p-file*, and *z-file*. The letter before the hyphen is called the tag. An auxiliary file name is formed from the SCCS file name: the last component of all SCCS file names must be of the form **s.***module-name*, the auxiliary files are named by replacing the leading **s** with the tag. The *g-file* is an exception to this scheme: the *g-file* is named by removing the **s.** prefix. For example, **s.xyz.c**, the auxiliary file names would be **xyz.c**, **l.xyz.c**, **p.xyz.c**, and **z.xyz.c**, respectively.

The *g-file*, which contains the generated text, is created in the current directory (unless the **–p** keyletter is used). A *g-file* is created in all cases, whether or not any lines of text were generated by the *get*. It is owned by the real user. If the **–k** keyletter is used or implied its mode is 644; otherwise its mode is 444. Only the real user need have write permission in the current directory.

The *l-file* contains a table showing which deltas were applied in generating the retrieved text. The *l-file* is created in the current directory if the **–l** keyletter is used; its mode is 444 and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the *l-file* have the following format:

a. A blank character if the delta was applied;
* otherwise.

b. A blank character if the delta was applied or was not applied and ignored;
* if the delta was not applied and was not ignored.

c. A code indicating a "special" reason why the delta was or was not applied:
"I": Included.
"X": Excluded.
"C": Cut off (by a **-c** keyletter).

d. Blank.

e. SCCS identification (SID).

f. Tab character.

g. Date and time (in the form YY/MM/DD HH:MM:SS) of creation.

h. Blank.

i. Login name of person who created *delta*.

The comments and MR data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

The *p-file* is used to pass information resulting from a *get* with an **-e** keyletter along to *delta*. Its contents are also used to prevent a subsequent execution of *get* with an **-e** keyletter for the same SID until *delta* is executed or the joint edit flag, **j**, [see *admin*(1)] is set in the SCCS file. The *p-file* is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. The format of the *p-file* is: the gotten SID, followed by a blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time the *get* was executed, followed by a blank and the **-i** keyletter argument if it was present, followed by a blank and the **-x** keyletter argument if it was present, followed by a new-line. There can be an arbitrary number of lines in the *p-file* at any time; no two lines can have the same new delta SID.

The *z-file* serves as a *lock-out* mechanism against simultaneous updates. Its contents are the binary (2 bytes) process ID of the command (i.e., *get*) that created it. The *z-file* is created in the directory containing the SCCS file for the duration of *get*. The same protection restrictions as those for the *p-file* apply for the *z-file*. The *z-file* is created mode 444.

FILES

g-file	Existed before the execution of *delta*; removed after completion of *delta*.
p-file	Existed before the execution of *delta*; may exist after completion of *delta*.
q-file	Created during the execution of *delta*; removed after completion of *delta*.
x-file	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z-file	Created during the execution of *delta*; removed during the execution of *delta*.
d-file	Created during the execution of *delta*; removed after completion of *delta*.
/usr/bin/bdiff	Program to compute differences between the "gotten" file and the *g-file*.

SEE ALSO

admin(1), delta(1), prs(1), what(1).
help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

If the effective user has write permission (either explicitly or implicitly) in the directory containing the SCCS files, but the real user does not, then only one file may be named when the **–e** keyletter is used.

NAME

infocmp – compare or print out terminfo descriptions

SYNOPSIS

infocmp [**–d**] [**–c**] [**–n**] [**–I**] [**–L**] [**–C**] [**–r**] [**–u**] [**–s d|i|l|c**] [**–v**] [**–V**] [**–1**] [**–w** width] [**–A** directory] [**–B** directory] [termname ...]

DESCRIPTION

infocmp can be used to compare a binary *terminfo*(4) entry with other terminfo entries, rewrite a *terminfo*(4) description to take advantage of the **use=** terminfo field, or print out a *terminfo*(4) description from the binary file (*term*(4)) in a variety of formats. In all cases, the boolean fields will be printed first, followed by the numeric fields, followed by the string fields.

Default Options

If no options are specified and zero or one *termnames* are specified, the **–I** option will be assumed. If more than one *termname* is specified, the **–d** option will be assumed.

Comparison Options [–d] [–c] [–n]

infocmp compares the *terminfo*(4) description of the first terminal *termname* with each of the descriptions given by the entries for the other terminal's *termnames*. If a capability is defined for only one of the terminals, the value returned will depend on the type of the capability: **F** for boolean variables, **–1** for integer variables, and **NULL** for string variables.

–d produce a list of each capability that is different. In this manner, if one has two entries for the same terminal or similar terminals, using *infocmp* will show what is different between the two entries. This is sometimes necessary when more than one person produces an entry for the same terminal and one wants to see what is different between the two.

–c produce a list of each capability that is common between the two entries. Capabilities that are not set are ignored. This option can be used as a quick check to see if the **–u** option is worth using.

–n produce a list of each capability that is in neither entry. If no *termnames* are given, the environment variable **TERM** will be used for both of the *termnames*. This can be used as a quick check to see if anything was left out of the description.

Source Listing Options [–I] [–L] [–C] [–r]

The **–I**, **–L**, and **–C** options will produce a source listing for each terminal named.

–I use the *terminfo*(4) names

–L use the long C variable name listed in **<term.h>**

–C use the *termcap* names

–r when using **–C**, put out all capabilities in *termcap* form

If no *termnames* are given, the environment variable **TERM** will be used for the terminal name.

The source produced by the **–C** option may be used directly as a *termcap* entry, but not all of the parameterized strings may be changed to the *termcap* format. *infocmp* will attempt to convert most of the parameterized information, but that which it doesn't will be plainly marked in the output and commented out. These should be edited by hand.

All padding information for strings will be collected together and placed at the beginning of the string where *termcap* expects it. Mandatory padding (padding information with a trailing '/') will become optional.

All *termcap* variables no longer supported by *terminfo*(4), but which are derivable from other *terminfo*(4) variables, will be output. Not all *terminfo*(4) capabilities will be translated; only those variables which were part of *termcap* will normally be output. Specifying the **–r** option will take off this restriction, allowing all capabilities to be output in *termcap* form.

Note that because padding is collected to the beginning of the capability, not all capabilities are output, mandatory padding is not supported, and *termcap* strings were not as flexible, it is not always possible to convert a *terminfo*(4) string capability into an equivalent *termcap* format. Not all of these strings will be able to be converted. A subsequent conversion of the *termcap* file back into *terminfo*(4) format will not necessarily reproduce the original *terminfo*(4) source.

Some common *terminfo* parameter sequences, their *termcap* equivalents, and some terminal types which commonly have such sequences, are:

Terminfo	Termcap	Representative Terminals
%p1%c	%.	adm
%p1%d	%d	hp, ANSI standard, vt100
%p1%'x'%+%c	%+x	concept
%i	%i	ANSI standard, vt100
%p1%?%'x'%>%t%p1%'y'%+%;	%>xy	concept
%p2 is printed before %p1	%r	hp

Use= Option [–u]

–u produce a *terminfo*(4) source description of the first terminal *termname* which is relative to the sum of the descriptions given by the entries for the other terminals *termnames*. It does this by analyzing the differences between the first *termname* and the other *termnames* and producing a description with **use=** fields for the other terminals. In this manner, it is possible to retrofit generic terminfo entries into a terminal's description. Or, if two similar terminals exist, but were coded at different times or by different people so that each description is a full description, using *infocmp* will show what can be done to change one description to be relative to the other.

A capability will get printed with an at-sign (@) if it no longer exists in the first *termname*, but one of the other *termname* entries contains a value for it. A capability's value gets printed if the value in the first *termname* is not found in any of the other *termname* entries, or if the first of the other *termname* entries that has this capability gives a different value for the capability than that in the first *termname*.

The order of the other *termname* entries is significant. Since the terminfo compiler **tic**(1M) does a left-to-right scan of the capabilities, specifying two **use=** entries that contain differing entries for the same capabilities will produce different results depending on the order that the entries are given in. *infocmp* will flag any such inconsistencies between the other *termname* entries as they are found.

Alternatively, specifying a capability *after* a **use=** entry that contains that capability will cause the second specification to be ignored. Using *infocmp* to recreate a description can be a useful check to make sure that everything was specified correctly in the original source description.

Another error that does not cause incorrect compiled files, but will slow down the compilation time, is specifying extra **use=** fields that are superfluous. *infocmp* will flag any other *termname* **use=** fields that were not needed.

Other Options [–s d|i|l|c] [–v] [–V] [–1] [–w width]

–s sort the fields within each type according to the argument below:

d leave fields in the order that they are stored in the *terminfo* database.

i sort by *terminfo* name.

l sort by the long C variable name.

c sort by the *termcap* name.

If no **–s** option is given, the fields printed out will be sorted alphabetically by the *terminfo* name within each type, except in the case of the **–C** or the **–L** options, which cause the sorting to be done by the *termcap* name or the long C variable name, respectively.

–v print out tracing information on standard error as the program runs.

–V print out the version of the program in use on standard error and exit.

–1 cause the fields to printed out one to a line. Otherwise, the fields will be printed several to a line to a maximum width of 60 characters.

–w change the output to width characters.

Changing Databases [–A directory] [–B directory]

The location of the compiled *terminfo*(4) database is taken from the environment variable **TERMINFO**. If the variable is not defined, or the terminal is not found in that location, the system *terminfo*(4) database, usually in */usr/lib/terminfo*, will be used. The options **–A** and **–B** may be used to override this location. The **–A** option will set **TERMINFO** for the first *termname* and the **–B** option will set **TERMINFO** for the other *termnames*. With this, it is possible to compare descriptions for a terminal with the same name located in two different databases. This is useful for comparing descriptions for the same terminal created by different people. Otherwise the terminals would have to be named differently in the *terminfo*(4) database for a comparison to be made.

FILES

/usr/lib/terminfo/?/* compiled terminal description database

DIAGNOSTICS

malloc is out of space!

There was not enough memory available to process all the terminal descriptions requested. Run *infocmp* several times, each time including a subset of the desired *termnames*.

use= order dependency found:

A value specified in one relative terminal specification was different from that in another relative terminal specification.

'use=*term*' did not add anything to the description.

A relative terminal name did not contribute anything to the final description.

must have at least two terminal names for a comparison to be done.

The **–u**, **–d** and **–c** options require at least two terminal names.

SEE ALSO

captoinfo(1M).
tic(1M), curses(3X), term(4), terminfo(4)

The **curses/terminfo** chapter in the *Programmer's Guide*.

NOTE

The *termcap* database (from earlier releases of SYSTEM V/68) may not be supplied in future releases.

NAME

install – install commands

SYNOPSIS

/etc/install [**–c** dira] [**–f** dirb] [**–i**] [**–n** dirc] [**–m** mode] [**–u** user] [**–g** group] [**–o**] [**–s**] file [dirx ...]

DESCRIPTION

The *install* command is most commonly used in "makefiles" [See *make*(1)] to install a *file* (updated target file) in a specific place within a file system. Each *file* is installed by copying it into the appropriate directory, thereby retaining the mode and owner of the original command. The program prints messages telling the user exactly what files it is replacing or creating and where they are going.

If no options or directories (*dirx* ...) are given, *install* will search a set of default directories (**/bin**, **/usr/bin**, **/etc**, **/lib**, and **/usr/lib**, in that order) for a file with the same name as *file*. When the first occurrence is found, *install* issues a message saying that it is overwriting that file with *file*, and proceeds to do so. If the file is not found, the program states this and exits without further action.

If one or more directories (*dirx* ...) are specified after *file*, those directories will be searched before the directories specified in the default list.

The meanings of the options are:

–c *dira* Installs a new command (*file*) in the directory specified by *dira*, only if it is not found. If it is found, *install* issues a message saying that the file already exists, and exits without overwriting it. May be used alone or with the **–s** option.

–f *dirb* Forces *file* to be installed in given directory, whether or not one already exists. If the file being installed does not already exist, the mode and owner of the new file will be set to **755** and **bin**, respectively. If the file already exists, the mode and owner will be that of the already existing file. May be used alone or with the **–o** or **–s** options.

–i Ignores default directory list, searching only through the given directories (*dirx ...*). May be used alone or with any other options except **–c** and **–f**.

–n *dirc* If *file* is not found in any of the searched directories, it is put in the directory specified in *dirc*. The mode and owner of the new file will be set to **755** and **bin**, respectively. May be used alone or with any other options except **–c** and **–f**.

–m *mode* The mode of the new file is set to *mode*. Only available to the superuser.

–u *user* The owner of the new file is set to *user*. Only available to the superuser.

–g *group* The group id of the new file is set to *group*. Only available to the superuser.

–o If *file* is found, this option saves the "found" file by copying it to **OLD***file* in the directory in which it was found. This option is useful when installing a frequently used file such as */bin/sh* or */etc/getty*, where the existing file cannot be removed. May be used alone or with any other options except **–c**.

–s Suppresses printing of messages other than error messages. May be used alone or with any other options.

SEE ALSO

make(1).

(1)

NAME

ld – link editor for common object files

SYNOPSIS

ld [options] filename

DESCRIPTION

The *ld* command combines several object files into one, performs relocation, resolves external symbols, and supports symbol table information for symbolic debugging. In the simplest case, the names of several object programs are given, and *ld* combines the objects, producing an object module that can either be executed or, if the **–r** option is specified, used as input for a subsequent *ld* run. 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, *filename*, is not an object file, *ld* assumes it is either an archive library or a text file containing link editor directives. [See *Link Editor Directives* in the *Programmer's Guide* for a discussion of input directives.]

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 library or a shared library. [See *Shared Libraries* in the *Programmer's Guide* for a discussion of shared libraries.] Only those routines defining an unresolved external reference are loaded. The library (archive) symbol table [see *ar*(4)] is searched sequentially with as many passes as are necessary to resolve external references which can be satisfied by library members. Thus, the ordering of library members is functionally unimportant, unless there exist multiple library members defining the same external symbol.

The following options are recognized by *ld*:

–e *epsym*
: Set the default entry point address for the output file to be that of the symbol *epsym*.

–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.

–l*x*
: Search a library **lib***x***.a**, where *x* is up to nine characters. A library is searched when its name is encountered, so the placement of a **–l** is significant. By default, libraries are located in *LIBDIR* or *LLIBDIR*.

–m Produce a map or listing of the input/output sections on the standard output.

–o *outfile*
Produce an output object file by the name *outfile*. The name of the default object file is **a.out**.

–r Retain relocation entries in the output object file. Relocation entries must be saved if the output file is to become an input file in a subsequent *ld* run. 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 *symname*
Enter *symname* as an undefined symbol in the symbol table. This is useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine. The placement of this option on the *ld* line is significant; it must be placed before the library which will define the symbol.

–x Do not preserve local symbols in the output symbol table; enter external and static symbols only. This option saves some space in the output file.

–z Do not bind anything to address zero. This option will allow runtime detection of null pointers.

–L *dir* Change the algorithm of searching for **lib***x***.a** to look in *dir* before looking in *LIBDIR* and *LLIBDIR*. This option is effective only if it precedes the **–l** option on the command line.

–M Output a message for each multiply-defined external definition.

–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.

–V Output a message giving information about the version of ld being used.

–VS *num*
Use *num* as a decimal version stamp identifying the **a.out** file that is produced. The version stamp is stored in the optional header.

–Y*[LU],dir*
Change the default directory used for finding libraries. If **L** is specified the first default directory which *ld* searches, *LIBDIR*, is replaced by *dir*. If **U** is specified and ld has been built with a second default directory, *LLIBDIR*, then that directory is replaced by *dir*. If ld was built with only one default directory and **U** is specified a warning is printed and the option is ignored.

FILES

LIBDIR/lib*x*.a	libraries
LLIBDIR/lib*x*.a	libraries
a.out	output file
LIBDIR	usually /lib
LLIBDIR	usually /usr/lib

SEE ALSO

as(1), cc(1), mkshlib(1), exit(2), end(3C), a.out(4), ar(4), and *Link Editor Directives* and *Shared Libraries* in the *Programmer's Guide.*

WARNING

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 insure the following properties for programs:

– C defines a zero pointer as null. A pointer to which zero has been assigned must not point to any object. To satisfy this, users must not place any object at virtual address zero in the 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( ) [see *exit*(2)] after execution of the main program. If the user calls the link editor directly, then the user must insure that the program always calls exit( ) rather than falling through the end of the entry routine.

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 complaining about "syntax errors".

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

NAME

lex – generate programs for simple lexical tasks

SYNOPSIS

lex [**–rctvn**] [file] ...

DESCRIPTION

The *lex* command generates programs to be used in simple lexical analysis of text.

The input *files* (standard input default) contain strings and expressions to be searched for, and C text to be executed when strings are found.

A file **lex.yy.c** is generated which, when loaded with the library, copies the input to the output except when a string specified in the file is found; then the corresponding program text is executed. The actual string matched is left in *yytext*, an external character array. Matching is done in order of the strings in the file. The strings may contain square brackets to indicate character classes, as in **[abx–z]** to indicate **a**, **b**, **x**, **y**, and **z**; and the operators *****, **+**, and **?** mean respectively any non-negative number of, any positive number of, and either zero or one occurrence of, the previous character or character class. The character **.** is the class of all ASCII characters except new-line. Parentheses for grouping and vertical bar for alternation are also supported. The notation *r*{*d*,*e*} in a rule indicates between *d* and *e* instances of regular expression *r*. It has higher precedence than |, but lower than *, ?, +, and concatenation. Thus **[a–zA–Z]+** matches a string of letters. The character ^ at the beginning of an expression permits a successful match only immediately after a new-line, and the character **$** at the end of an expression requires a trailing new-line. The character / in an expression indicates trailing context; only the part of the expression up to the slash is returned in *yytext*, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within " symbols or preceded by \.

Three subroutines defined as macros are expected: **input()** to read a character; **unput(***c***)** to replace a character read; and **output(***c***)** to place an output character. They are defined in terms of the standard streams, but you can override them. The program generated is named **yylex()**, and the library contains a **main()** which calls it. The action REJECT on the right side of the rule causes this match to be rejected and the next suitable match executed; the function **yymore()** accumulates additional characters into the same *yytext*; and the function **yyless(***p***)** pushes back the portion of

the string matched beginning at *p*, which should be between *yytext* and *yytext+yyleng*. The macros *input* and *output* use files **yyin** and **yyout** to read from and write to, defaulted to **stdin** and **stdout**, respectively.

Any line beginning with a blank is assumed to contain only C text and is copied; if it precedes **%%** it is copied into the external definition area of the **lex.yy.c** file. All rules should follow a **%%**, as in YACC. Lines preceding **%%** which begin with a non-blank character define the string on the left to be the remainder of the line; it can be called out later by surrounding it with **{}**. Note that curly brackets do not imply parentheses; only string substitution is done.

EXAMPLE

```
D       [0–9]
%%
if      printf("IF statement\n");
[a–z]+  printf("tag, value %s\n",yytext);
0{D}+   printf("octal number %s\n",yytext);
{D}+    printf("decimal number %s\n",yytext);
"++"    printf("unary op\n");
"+"     printf("binary op\n");
"/*"     skipcommnts();
%%
 skipcommnts()
 {
        for (;;)
        {
                while (input() != '*')
                        ;
                if (input() != '/')
                        unput(yytext[yyleng-1]);
                else
                        return;
        }
 }
```

The external names generated by *lex* all begin with the prefix **yy** or **YY**.

The flags must appear before any files. The flag **–r** indicates RATFOR actions, **–c** indicates C actions and is the default, **–t** causes the **lex.yy.c** program to be written instead to standard output, **–v** provides a one-line summary of statistics, **–n** will not print out the **–v** summary. Multiple files are treated as a single file. If no files are specified, standard input is

used.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

%p *n* number of positions is *n* (default 2500)

%n *n* number of states is *n* (500)

%e *n* number of parse tree nodes is *n* (1000)

%a *n* number of transitions is *n* (2000)

%k *n* number of packed character classes is *n* (1000)

%o *n* size of output array is *n* (3000)

The use of one or more of the above automatically implies the **–v** option, unless the **–n** option is used.

SEE ALSO

yacc(1).
The chapter, "yacc," in the *Programmer's Guide.*

BUGS

The **–r** option is not yet fully operational.

NAME

lint – a C program checker

SYNOPSIS

lint [option] ... file ...

DESCRIPTION

The *lint* command attempts to detect features of the C program files that are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compilers. Among the things that are currently detected are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions that return values in some places and not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but none returned.

Arguments whose names end with **.c** are taken to be C source files. Arguments whose names end with **.ln** are taken to be the result of an earlier invocation of *lint* with either the **–c** or the **–o** option used. The **.ln** files are analogous to **.o** (object) files that are produced by the *cc*(1) command when given a **.c** file as input. Files with other suffixes are warned about and ignored.

lint will take all the **.c**, **.ln**, and **llib-l***x***.ln** (specified by **–l***x*) files and process them in their command line order. By default, *lint* appends the standard C lint library (**llib-lc.ln**) to the end of the list of files. However, if the **–p** option is used, the portable C lint library (**llib-port.ln**) is appended instead. When the **–c** option is not used, the second pass of *lint* checks this list of files for mutual compatibility. When the **–c** option is used, the **.ln** and the **llib-l***x***.ln** files are ignored.

Any number of *lint* options may be used, in any order, intermixed with file-name arguments. The following options are used to suppress certain kinds of complaints:

–a Suppress complaints about assignments of long values to variables that are not long.

–b Suppress complaints about **break** statements that cannot be reached. (Programs produced by *lex* or *yacc* will often result in many such complaints).

–h Do not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

–u Suppress complaints about functions and external variables used and not defined, or defined and not used. (This option is suitable for running *lint* on a subset of files of a larger program).

–v Suppress complaints about unused arguments in functions.

–x Do not report variables referred to by external declarations but never used.

The following arguments alter *lint*'s behavior:

–l*x* Include additional lint library **llib-l***x***.ln**. For example, you can include a lint version of the math library **llib-lm.ln** by inserting **–lm** on the command line. This argument does not suppress the default use of **llib-lc.ln**. These lint libraries must be in the assumed directory. This option can be used to reference local lint libraries and is useful in the development of multi-file projects.

–n Do not check compatibility against either the standard or the portable lint library.

–p Attempt to check portability to other dialects (IBM and GCOS) of C. Along with stricter checking, this option causes all non-external names to be truncated to eight characters and all external names to be truncated to six characters and one case.

–c Cause *lint* to produce a **.ln** file for every **.c** file on the command line. These **.ln** files are the product of *lint*'s first pass only, and are not checked for inter-function compatibility.

–o lib Cause *lint* to create a lint library with the name **llib-l***lib***.ln**. The **–c** option nullifies any use of the **–o** option. The lint library produced is the input that is given to *lint*'s second pass. The **–o** option simply causes this file to be saved in the named lint library. To produce a **llib-l***lib***.ln** without extraneous messages, use of the **–x** option is suggested. The **–v** option is useful if the source file(s) for the lint library are just external interfaces (for example, the way the file **llib-lc** is written). These option settings are also available through the use of "lint comments" (see below).

The **–D**, **–U**, and **–I** options of *cpp*(1) and the **–g** and **–O** options of *cc*(1) are also recognized as separate arguments. The **–g** and **–O** options are ignored, but, by recognizing these options, *lint*'s behavior is closer to that

of the *cc*(1) command. Other options are warned about and ignored. The pre-processor symbol "lint" is defined to allow certain questionable code to be altered or removed for *lint*. Therefore, the symbol "lint" should be thought of as a reserved word for all code that is planned to be checked by *lint*.

Certain conventional comments in the C source will change the behavior of *lint*:

/*NOTREACHED*/
: at appropriate points stops comments about unreachable code. [This comment is typically placed just after calls to functions like *exit*(2)].

/*VARARGS*n*/
: suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/*ARGSUSED*/
: turns on the **-v** option for the next function.

/*LINTLIBRARY*/
: at the beginning of a file shuts off complaints about unused functions and function arguments in this file. This is equivalent to using the **-v** and **-x** options.

lint produces its first output on a per-source-file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, if the **-c** option is not used, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source file name will be printed followed by a question mark.

The behavior of the **-c** and the **-o** options allows for incremental use of *lint* on a set of C source files. Generally, one invokes *lint* once for each source file with the **-c** option. Each of these invocations produces a **.ln** file which corresponds to the **.c** file, and prints all messages that are about just that source file. After all the source files have been separately run through *lint*, it is invoked once more (without the **-c** option), listing all the **.ln** files with the needed **-l***x* options. This will print all the inter-file

inconsistencies. This scheme works well with *make*(1); it allows *make* to be used to *lint* only the source files that have been modified since the last time the set of source files were *lint*ed.

FILES

LLIBDIR	the directory where the lint libraries specified by the –l*x* option must exist, usually /usr/lib
LLIBDIR/lint[12]	first and second passes
LLIBDIR/llib-lc.ln	declarations for C Library functions (binary format; source is in *LLIBDIR*/llib-lc)
LLIBDIR/llib-port.ln	declarations for portable functions (binary format; source is in *LLIBDIR*/llib-port)
LLIBDIR/llib-lm.ln	declarations for Math Library functions (binary format; source is in *LLIBDIR*/llib-lm)
TMPDIR/*lint*	temporaries
TMPDIR	usually /usr/tmp but can be redefined by setting the environment variable **TMPDIR** [see *tempnam*() in *tmpnam*(3S)].

SEE ALSO

cc(1), cpp(1), make(1).

BUGS

exit(2), *setjmp*(3C), and other functions that do not return are not understood; this causes various lies.

NAME

list – produce C source listing from a common object file

SYNOPSIS

list [–V] [–h] [–F function] source-file . . . [object-file]

DESCRIPTION

The *list* command produces a C source listing with line number information attached. If multiple C source files were used to create the object file, *list* will accept multiple file names. The object file is taken to be the last non-C source file argument. If no object file is specified, the default object file, **a.out**, will be used.

Line numbers will be printed for each line marked as breakpoint inserted by the compiler (generally, each executable C statement that begins a new line of source). Line numbering begins anew for each function. Line number 1 is always the line containing the left curly brace ({) that begins the function body. Line numbers will also be supplied for inner block redeclarations of local variables so that they can be distinguished by the symbolic debugger.

The following options are interpreted by *list* and may be given in any order:

–V Print, on standard error, the version number of the *list* command executing.

–h Suppress heading output.

–F*function* List only the named function. The **–F** option may be specified multiple times on the command line.

SEE ALSO

as(1), cc(1), ld(1).

WARNING

Object files given to *list* must have been compiled with the **–g** option of *cc*(1).

Since *list* does not use the C preprocessor, it may be unable to recognize function definitions whose syntax has been distorted by the use of C preprocessor macro substitutions.

DIAGNOSTICS

list will produce the error message "list: name: cannot open" if *name* cannot be read. If the source file names do not end in **.c** , the message is "list: name: invalid C source name". An invalid object file will cause the

message "list: name: bad magic" to be produced.

If some or all of the symbolic debugging information is missing, one of the following messages will be printed:
"list: name: symbols have been stripped, cannot proceed",
"list: name: cannot read line numbers", and
"list: name: not in symbol table".

The following messages are produced when *list* has become confused by **#ifdef's** in the source file:
"list: name: cannot find function in symbol table",
"list: name: out of sync: too many }", and
"list: name: unexpected end-of-file".

The error message "list: name: missing or inappropriate line numbers" means that either symbol debugging information is missing, or *list* has been confused by C preprocessor statements.

NAME

lorder – find ordering relation for an object library

SYNOPSIS

lorder file ...

DESCRIPTION

The input is one or more object or library archive *files* [see *ar*(1)]. The standard output is a list of pairs of object file or archive member names, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by *tsort*(1) to find an ordering of a library suitable for one-pass access by *ld*(1). Note that the link editor *ld*(1) is capable of multiple passes over an archive in the portable archive format [see *ar*(4)] and does not require that *lorder*(1) be used when building an archive. The usage of the *lorder*(1) command may, however, allow for a slightly more efficient access of the archive during the link edit process.

The following example builds a new library from existing **.o** files.

```
ar –cr library `lorder *.o | tsort`
```

FILES

TMPDIR/*symref	temporary files
TMPDIR/*symdef	temporary files

TMPDIR is usually /usr/tmp but can be redefined by setting the environment variable **TMPDIR** [see *tempnam*( ) in *tmpnam*(3S)].

SEE ALSO

ar(1), ld(1), tsort(1), ar(4).

WARNING

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

The *m4* command is a macro processor intended as a front end for Ratfor, 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 4,096.

–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 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 SYSTEM V/68 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 SYSTEM V/68 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 XXXXX in its argument with 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; 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*.

SEE ALSO

cc(1), cpp(1).

NAME

make – maintain, update, and regenerate groups of programs

SYNOPSIS

make [**–f** makefile] [**–p**] [**–i**] [**–k**] [**–s**] [**–r**] [**–n**] [**–b**] [**–e**] [**–u**] [**–t**] [**–q**] [names]

DESCRIPTION

The *make* command allows the programmer to maintain, update, and regenerate groups of computer programs. The following is a brief description of all options and some special names:

–f *makefile* Description file name. *makefile* is assumed to be the name of a description file.

–p Print out the complete set of macro definitions and target descriptions.

–i Ignore error codes returned by invoked commands. This mode is entered if the fake target name **.IGNORE** appears in the description file.

–k Abandon work on the current entry if it fails, but continue on other branches that do not depend on that entry.

–s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name **.SILENT** appears in the description file.

–r Do not use the built-in rules.

–n No execute mode. Print commands, but do not execute them. Even lines beginning with an @ are printed.

–b Compatibility mode for old makefiles.

–e Environment variables override assignments within makefiles.

–u Force an unconditional update.

–t Touch the target files (causing them to be up-to-date) rather than issue the usual commands.

–q Question. The *make* command returns a zero or non-zero status code depending on whether the target file is or is not up-to-date.

.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.

.PRECIOUS Dependents of this target will not be removed when quit or interrupt are hit.

.SILENT Same effect as the **–s** option.

.IGNORE Same effect as the **–i** option.

make executes commands in *makefile* to update one or more target *names*. *Name* is typically a program. If no **–f** option is present, **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 **–** *makefile* argument pair may appear.

make updates a target only if its dependents are newer than the target (unless the **–u** option is used to force an unconditional update). All prerequisite files of a target are added recursively to the list of targets. Missing files are deemed to be out-of-date.

makefile 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 the <backslash><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.

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 */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 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 discussion of the **MAKEFLAGS** macro under *Environment*). 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, or the entry **.IGNORE:** appears in *makefile*, or 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.

The **–b** option allows old makefiles (those written for the old version of *make*) to run without errors.

Interrupt and quit cause the target to be deleted unless the target is a dependent of the special name **.PRECIOUS**.

Environment

The environment is read by *make*. All variables are assumed to be macro definitions and 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 proves very useful for "super-makes". In fact, as noted above, when the **–n** option is used, the command **$(MAKE)** is executed anyway; hence, one can perform a **make –n** recursively on a whole software system to see what would have been executed. This is because the **–n** is put in **MAKEFLAGS** and passed to further invocations of **$(MAKE)**. This 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 file name 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 under *Libraries*.

Internal Macros

There are five internally maintained macros which are useful for writing rules for building targets.

$*** The macro **$* stands for the file name 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 which is out-of-date with respect to the target (i.e., 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 out-of-date with respect to the target; essentially, those modules which 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 which 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~ .sh .sh~
.c.o .c.a .c~.o .c~.c .c~.a
.f.o .f.a .f~.o .f~.f .f~.a
.h~.h .s.o .s~.o .s~.s .s~.a .sh~.sh
.l.o .l.c .l~.o .l~.l .l~.c
.y.o .y.c .y~.o .y~.y .y~.c
```

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 –fp – 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 *make*'s 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 (i.e., **.c:**) is the definition of how to build *x* from *x*.**c**. In effect, the other suffix is null. This is useful for building targets from only one source file (e.g., shell procedures, 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~

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 is 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 **$(LIB)(file.o)** both refer to an archive library which contains **file.o**. (This assumes the **LIB** macro has been previously defined.) The expression **$(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 byproduct 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 file names (inside **lib**) whose C source files are out-of-date. The substitution mode translates the **.o** to **.c**. (Unfortunately, one cannot as yet transform to **.c**~; however, this may become possible in the future.) Note also, 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
/bin/sh

SEE ALSO

cc(1), lex(1), yacc(1), printf(3S), sccsfile(4).
cd(1), sh(1) in the *User's Reference Manual.*

NOTES

Some commands return non-zero status inappropriately; use **-i** to overcome the difficulty.

BUGS

File names with the characters = : @ will not work. Commands that are directly executed by the shell, notably *cd*(1), are ineffectual across newlines in *make*. The syntax **(lib(file1.o file2.o file3.o)** is illegal. You cannot build **lib(file.o)** from **file.o**. The macro **$(a:.o=.c~)** does not work. Named pipes are not handled well.

NAME

mcs – manipulate the object file comment section

SYNOPSIS

mcs [options] object-file ...

DESCRIPTION

The *mcs* command manipulates the comment section, normally the ".comment" section, in an object file. It is used to add to, delete, print, and compress the contents of the comment section in a SYSTEM V/68 object file. *mcs* must be given one or more of the options described below. It takes each of the options given and applies them in order to the *object-files*.

If the object file is an archive, the file is treated as a set of individual object files. For example, if the **–a** option is specified, the string is appended to the comment section of each archive element.

The following options are available.

–a *string*
: Append *string* to the comment section of the *object-files*. If *string* contains embedded blanks, it must be enclosed in quotation marks.

–c
: Compress the contents of the comment section. All duplicate entries are removed. The ordering of the remaining entries is not disturbed.

–d
: Delete the contents of the comment section from the object file. The object file comment section header is removed also.

–n *name*
: Specify the name of the section to access. 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. If more than one name is specified, each entry printed is tagged by the name of the file from which it was extracted, using the format "filename:string."

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/*	temporary files

TMPDIR is usually /usr/tmp but can be redefined by setting the environment variable **TMPDIR** [see *tempnam*( ) in *tmpnam*(3S)].

SEE ALSO

cpp(1), a.out(4).

NAME

mkmenus – extracts menus from labels stored in command shell scripts

SYNOPSIS

mkmenus menudirectory pid ...

DESCRIPTION

mkmenus generates menus, reads the user's response to these menus, then generates the next menu or performs the function listed in the menu. Features include menu headings, access to help information, and the ability to go back to a previous menu.

mkmenus permits sub-menus and sub-commands to be added and deleted easily. The menu hierarchy is defined or modified simply by placing command shell scripts in appropriate directories and inserting menu labels in the shell scripts. Subcommands "stubs" (place holders for as yet undelivered functions) are provided by setting up command shell scripts that contain only menu information. Code is easy to locate because the menu structure and file structure are identical.

Invocation

The *menudirectory* argument specifies the starting menu directory. The user of *mkmenus* does not provide the argument *pid*. This argument is passed by *mkmenus* to itself recursively. It is used to terminate earlier invocations of *mkmenus* when the user wishes to quit the entire menu system.

File Naming Conventions

In the *menudirectory*, and every directory under it, *mkmenus* expects to find a file named DESC. DESC files provide menu information for directories through the menu labels discussed below.

Other files in the directories are taken as sub-commands, provided they have the appropriate menu labels. The file name becomes the menu name of the sub-command.

Directories without DESC files and files without menu labels are quietly ignored.

Menu Labels

All menu labels are of the form #<word># and must be at the beginning of a line.

mkmenus searches the current directory for files (and directories with DESC files) for lines with #menu# labels. The labels are removed and the remainder of the line is placed in a menu. Files can contain only one

#menu# line. Tabs and spaces following **#menu#** labels are ignored. Any shell scripts that do not have **#menu#** labels will not appear in a menu. *mkmenus* automatically numbers each menu item.

Heading Labels

Lines with **#head#** labels are used to add headings to menus. These lines are only meaningful when placed in DESC files. Headings may contain as many lines as desired, allowing the developer to place an introduction above a menu as shown in the DESC file below:

```
#head#      ED-GAMES MENU
#head#
#head# These games are designed to teach children basic math
#head# and reading skills. They work only on a CRT.

#menu# educational games
```

Help Labels

Lines with **#help#** labels are used by *mkmenus* to provide users with help information. Users request help by typing **?** (question mark) alone on a line or **n?** where **n** is a menu item number. The **?** alone is meant to produce a summary description of all commands in one menu and is taken from the DESC file. The **n?** response is meant to produce help specific to a single menu item and is taken from the appropriate shell file. *mkmenus* will print the message "No help available" if it cannot find **#help#** labels.

Returning to Previous Menu

A **^** (circumflex) in response to the prompt causes *mkmenus* to "go back" to the previous menu. There is no **^** available at the top-level menu.

Leaving the Menu System

Users can exit the *mkmenus* menu system by typing **q** at the "Enter a number" prompt.

General

Labels can be placed anywhere in a DESC or shell script file. The order of the 3 different types of labels does not matter. If a sub-directory accessed by *mkmenus* contains no **#menu#** labeled files, the message "{directory} (No functions available yet)" is printed and *mkmenus* terminates with an exit code if **1**.

In addition to generating menus, *mkmenus* prompts the user for a numeric response, validates this response, and then determines what to do next: print help information, print next menu, or execute a command.

EXAMPLES

Shell Script Format

A command shell script for *cat* might read:

```
#menu# prints files
#help# The cat command concatenates and prints files.
#help# After prompting you for file names, cat prints
#help# the contents of the files at your terminal.

        <<code to execute cat command>>
```

Directory Structure

A typical directory layout for use by *mkmenus* is shown below. Names marked with * are directories. Each indentation indicates the next lower level of directories.

```
menudirectory*
        DESC
        machinemgmt*
           DESC
             :
             :
        softwaremgmt*
           DESC
             :
             :
        syssetup*
           DESC
           datetime
           nodename
        userserv*
           DESC
           addgroup
           adduser
           deluser
           moduser*
                DESC
                chgloginid
                chgpasswd
                chgshell
```

Sample Menu

A typical menu generated by *mkmenus* would look like:

```
                SYSTEM ADMINISTRATION
1 machinemgmt     machine management menu
2 softwaremgmt    software management menu
3 syssetup        system setup menu
4 userserv        user services menu
Enter a number or ...
? or <number>? for HELP, q to QUIT:
```

BUGS

If a file contains more than one **#menu#** line, the extra lines are quietly ignored.

If a file contains more than 22 help lines, the top lines may scroll off the user's screen (depending on the terminal type).

SEE ALSO

sysadm(1).

(1)

NAME

nm – print name list of common object file

SYNOPSIS

nm [–oxhvnefurpVT] filename ...

DESCRIPTION

The *nm* command displays the symbol table of each common object file, *filename*. *Filename* may be a relocatable or absolute common object file; or it may be an archive of relocatable or absolute common object files. For each symbol, the following information will be printed:

Name The name of the symbol.

Value Its value expressed as an offset or an address depending on its storage class.

Class Its storage class.

Type Its 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 object file must have been compiled with the **–g** option of the *cc*(1) command for this information to appear.

Size Its size in bytes, if available. Note that the object 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 it is defined, if available. Note that the object 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).

The output of *nm* may be controlled using the following options:

–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.

–e Print only external and static symbols.

–f Produce full output. Print redundant symbols (.text, .data, .lib, and .bss), normally suppressed.

–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 segment symbol), **D** (data segment symbol), **S** (user defined segment symbol), **R** (register symbol), **F** (file symbol), or **C** (common symbol). If the symbol is local (non-external), the type letter is in lower case.

–V Print the version of the nm command executing on the standard error output.

–T By default, *nm* prints the entire name of the symbols listed. Since object files can have symbols 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 *tempnam*( ) in *tmpnam*(3S)].

BUGS

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.

SEE ALSO

as(1), cc(1), ld(1), tmpnam(3S), a.out(4), ar(4).

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*.

(1)

NAME

prof – display profile data

SYNOPSIS

prof [**–tcan**] [**–ox**] [**–g**] [**–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 external 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 printing of 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 following options may be used in any combination:

–g Include non-global symbols (static functions).

–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.

A program creates a profile file if it has been loaded 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 when the file containing the function was compiled.

The name of the file created by a profiled program is controlled by the environment variable PROFDIR. If PROFDIR does not exist, "mon.out" is produced in the directory that is current when the program terminates. If PROFDIR = string, "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 for profile
a.out for namelist

SEE ALSO

cc(1), exit(2), profil(2), monitor(3C), prof(5).

WARNING

The times reported in successive identical runs may show variances of 20% or more, because of varying cache-hit ratios due to sharing of 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.

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, i.e., the static function call counts are not added in with the call counts of the external function.

WARNING

Only programs that call *exit*(2) or return from *main* will cause a profile file to be produced, unless a final call to monitor is explicitly coded.

The use of the **-p** option to *cc*(1) to invoke profiling imposes a limit of 600 functions that may have call counters established during program execution. For more counters you must call *monitor*(3C) directly. If this limit is exceeded, other data will be overwritten and the **mon.out** file will be corrupted. The number of call counters used will be reported automatically by the *prof* command whenever the number exceeds 5/6 of the maximum.

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* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**), and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file 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]* Used to specify the *SCCS* ID*entification* (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.

–e Requests information for all deltas created *earlier* than and including the delta designated via the **–r** keyletter or the date given by the **–c** option.

–l Requests information for all deltas created *later* than and including the delta designated via the **–r** keyletter or the date given by the **–c** option.

c date-time The cutoff date-time **–c**[cutoff]] is 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"**.

–a Requests 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. There is no limit on the number of times a data keyword may appear in a *dataspec*.

The information printed by *prs* consists of: (1) the user-supplied text; and (2) appropriate values (extracted from the SCCS file) substituted for the recognized data keywords in the order of appearance in the *dataspec*. The format of a data keyword value is either *Simple* (S), in which keyword substitution is direct, or *Multi-line* (M), 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

TABLE 1. SCCS Files Data Keywords (continued)

Keyword	*Data Item*	*File Section*	*Value*	*Format*
: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?????

SEE ALSO

admin(1), delta(1), get(1), sccsfile(4).
help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

regcmp – regular expression compile

SYNOPSIS

regcmp [–] files

DESCRIPTION

The *regcmp* command performs a function similar to *regcmp*(3X) and, in most cases, precludes the need for calling *regcmp*(3X) from C programs. This 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 will be placed in *file*.**c**. The format of entries in *file* is a name (C variable) followed by one or more blanks followed by a regular expression 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 which uses the *regcmp* output, *regex*(*abc*,*line*) will apply 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"

In the C program that uses the *regcmp* output,

regex(telno, line, area, exch, rest)

will apply the regular expression named *telno* to *line*.

SEE ALSO

regcmp(3X).

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 newest (most recent) delta in its branch in the delta chain of each named SCCS file. In addition, the specified must *not* be that of a version being edited for the purpose of making a delta (i. e., if a *p-file* [see *get*(1)] exists for the named SCCS file, the specified must *not* appear in any entry of the *p-file*).

The **–r** option is used for specifying the *SID* (SCCS IDentification) level of the delta to be removed.

If a directory is named, *rmdel* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored.

Simply stated, they are either (1) if you make a delta you can remove it; or (2) if you own the file and directory you can remove a delta.

FILES

x.file [see *delta*(1)]
z.file [see *delta*(1)]

SEE ALSO

delta(1), get(1), prs(1), sccsfile(4).
help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

sact – print current SCCS file editing activity

SYNOPSIS

sact files

DESCRIPTION

sact informs the user of any impending deltas to a named SCCS file. This situation occurs when *get*(1) with the **–e** option has been previously executed without a subsequent execution of *delta*(1). If a directory is named on the command line, *sact* behaves as though each file in the directory were specified 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 being taken as the name of an SCCS file to be processed.

The output for each named file consists of five fields separated by spaces.

Field 1	specifies the SID of a delta that currently exists in the SCCS file to which changes will be made to make the new delta.
Field 2	specifies the SID for the new delta to be created.
Field 3	contains the logname of the user who will make the delta (i.e., executed a *get* for editing).
Field 4	contains the date that **get –e** was executed.
Field 5	contains the time that **get –e** was executed.

SEE ALSO

delta(1), get(1), unget(1).

DIAGNOSTICS

Use *help*(1) for explanations.

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 the two versions. 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).
–**s***n*	*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.

FILES

/tmp/get????? Temporary files

SEE ALSO

get(1).
bdiff(1), help(1), pr(1) in the *User's Reference Manual.*

DIAGNOSTICS

"*file*: No differences" If the two versions are the same.
Use *help*(1) for explanations.

NAME

sdb – symbolic debugger

SYNOPSIS

sdb [–w] [–W] [objfil [corfil [directory-list]]]

DESCRIPTION

The *sdb* command calls a symbolic debugger that can be used with C and F77 programs. It may be used to examine their object files and core files and to provide a controlled environment for their execution.

Objfil is an executable program file which has been 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 default for *objfil* is **a.out**. *Corfil* is assumed to be a core image file produced after executing *objfil*; the default for *corfil* is **core**. The core file need not be present. A – in place of *corfil* will force *sdb* to ignore any core image file. The colon separated list of directories (*directory-list*) is used to locate the source files used to build *objfil*.

It is useful to know that at any time there is a *current line* and *current file*. If *corfil* exists then they are initially set to the line and file containing the source statement at which the process terminated. Otherwise, they are set to the first line in *main()*. The current line and file may be changed with the source file examination commands.

By default, warnings are provided if the source files used in producing *objfil* cannot be found, or are newer than *objfil*. This checking feature and the accompanying warnings may be disabled by the use of the **–W** flag.

Names of variables are written just as they are in C or F77. *sdb* does not truncate names. 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.

It is also possible to refer to structure members as *variable.member*, pointers to structure members as *variable–>member* and array elements as *variable*[*number*]. Pointers may be dereferenced by using the form *pointer*[0]. Combinations of these forms may also be used. F77 common variables may be referenced by using the name of the common block instead of the structure name. Blank common variables may be named by the form *.variable*. A number may be used in place of a structure variable name, in which case the number is viewed as the address of the structure, and the template used for the structure is that of the last structure

referenced by *sdb*. An unqualified structure variable may also be used with various commands. Generally, *sdb* will interpret a structure as a set of variables. Thus, *sdb* will display the values of all the elements of a structure when it is requested to display a structure. An exception to this interpretation occurs when displaying variable addresses. An entire structure does have an address, and it is this value *sdb* displays, not the addresses of individual elements.

Elements of a multidimensional array may be referenced 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. As with structures, *sdb* displays all the values of an array or of the section of an array if trailing subscripts are omitted. It displays only the address of the array itself or of the section specified by the user if subscripts are omitted. A multidimensional parameter in an F77 program cannot be displayed as an array, but it is actually a pointer, whose value is the location of the array. The array itself can be accessed symbolically from the calling function.

A particular instance of a variable on the stack may be referenced by using the form *procedure:variable,number*. All the variations mentioned in naming variables may be used. *Number* is the occurrence of the specified procedure on the stack, counting the top, or most current, as the first. If no procedure is specified, the procedure currently executing is used by default.

It is also possible to specify a variable by its address. All forms of integer constants which are valid in C may be used, so that addresses may be input in decimal, octal or hexadecimal.

Line numbers in the source program are referred to as *file-name:number* or *procedure:number*. In either case the number is relative to the beginning of the file. If no procedure or file name is given, the current file is used by default. If no number is given, the first line of the named procedure or file is used.

While a process is running under *sdb*, all addresses refer to the executing program; otherwise they refer to *objfil* or *corfil*. An initial argument of **–w** permits overwriting locations in *objfil*.

Addresses

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by two triples (*b1*, *e1*, *f1*) and (*b2*, *e2*, *f2*) and the *file address* corresponding to a written *address* is calculated as follows:

b1<=*address*<*e1*

file address=*address*+*f1*–*b1*

otherwise

b2<=*address*<*e2*

file address=*address*+*f2*–*b2*,

otherwise, the requested *address* is not legal. In some cases (e.g., for programs with separated I and D space) the two segments for a file may overlap.

The initial setting of both mappings is suitable for normal **a.out** and **core** files. If either file is not of the kind expected then, for that file, *b1* is set to 0, *e1* is set to the maximum file size, and *f1* is set to 0; in this way the whole file can be examined with no address translation.

In order for *sdb* to be used on large files, all appropriate values are kept as signed 32-bit integers.

Commands

The commands for examining data in the program are:

t Print a stack trace of the terminated or halted program.

T Print the top line of the stack trace.

variable/*clm*

Print the value of *variable* according to length *l* and format *m*. A 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 decimal
u decimal, unsigned
o octal
x hexadecimal
f 32-bit single precision floating point
g 64-bit double precision floating point
s Assume *variable* is a string pointer and print characters starting at the address pointed to by the variable.
a Print characters starting at the variable's address. This format may not be used with register variables.
p pointer to procedure
i disassemble machine-language instruction with addresses printed numerically and symbolically.
I disassemble machine-language instruction with addresses just printed numerically.

Length specifiers are only effective with the **c**, **d**, **u**, **o** and **x** formats. Any of the specifiers, *c*, *l*, and *m*, may be omitted. If all are omitted, *sdb* chooses a length and a format suitable for the variable's type as declared in the program. If *m* is specified, then this format is used for displaying the variable. A length specifier determines the output length of the value to be displayed, sometimes resulting in truncation. A count specifier *c* tells *sdb* to display that many units of memory, beginning at the address of *variable*. The number of bytes in one such unit of memory is determined by the length specifier *l*, or if no length is given, by the size associated with the *variable*. If a count specifier is used for 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 **./**.

The *sh*(1) metacharacters * and ? may be used within procedure and variable names, providing a limited form of pattern matching. If no procedure name is given, variables local to the current procedure and global variables are matched; if a procedure name is specified then only variables local to that procedure are matched. To match only global variables, the form *:pattern* is used.

linenumber?*lm*
variable:?*lm*
Print the value at the address from **a.out** or I space given by *linenumber* or *variable* (procedure name), according to the format *lm*. The default format is 'i'.

variable=*lm*
linenumber=*lm*
number=*lm*
Print the address of *variable* or *linenumber*, or the value of *number*, in the format specified by *lm*. If no format is given, then **lx** is used. The last variant of this command provides a convenient way to convert between decimal, octal and hexadecimal.

variable!*value*
Set *variable* to the given *value*. The value may be a number, a character constant or a variable. The value must be well defined; expressions which produce more than one value, such as structures, are not allowed. 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 are viewed as integers. The *variable* may be an expression which indicates more than one variable, such as an array or structure name. 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.

x Print the machine registers and the current machine-language instruction.

X Print the current machine-language instruction.

The commands for examining source files are:

e *procedure*
e *file-name*
e *directory/*
e *directory file-name*
The first two forms set the current file to the file containing *procedure* or to *file-name*. The current line is set to the first line in the named procedure or file. Source files are assumed to be in *directory*. The default is the current working directory. The latter two forms change the value of *directory*. If no procedure, file name, or directory is given, the current procedure name and file name are

reported.

/regular expression/
Search forward from the current line for a line containing a string matching *regular expression* as in *ed*(1). The trailing / may be deleted.

?regular expression?
Search backward from the current line for a line containing a string matching *regular expression* as in *ed*(1). The trailing ? may be deleted.

p Print the current line.

z Print the current line followed by the next 9 lines. Set the current line to the last line printed.

w Window. Print the 10 lines around the current line.

number
Set the current line to the given line number. Print the new current line.

count **+**
Advance the current line by *count* lines. Print the new current line.

count **−**
Retreat the current line by *count* lines. Print the new current line.

The commands for controlling the execution of the source program are:

count **r** *args*
count **R**
Run the program with the given arguments. The **r** command with no arguments reuses the previous arguments to the program while the **R** command runs the program with no arguments. An argument beginning with < or > causes redirection for the standard input or output, respectively. If *count* is given, it specifies the number of breakpoints to be ignored.

linenumber **c** *count*
linenumber **C** *count*
Continue after a breakpoint or interrupt. If *count* is given, the program will stop when *count* breakpoints have been encountered. The signal which caused the program to stop 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 is continued. The breakpoint is deleted when the command finishes.

linenumber **g** *count*
Continue after a breakpoint with execution resumed at the given line. If *count* is given, it specifies the number of breakpoints to be ignored.

s *count*
S *count*
Single step the program through *count* lines. If no count is given then the program is run for one line. **S** is equivalent to **s** except it steps through procedure calls.

i
I Single step by one machine-language instruction. The signal which caused the program to stop is reactivated with the **I** command and ignored with the **i** command.

variable$**m** *count*
address:**m** *count*
Single step (as with **s**) until the specified location is modified with a new value. If *count* is omitted, it is effectively infinity. *Variable* must be accessible from the current procedure. Since this command is done by software, it can be very slow.

level **v**
Toggle verbose mode, for use when single stepping with **S**, **s** or **m**. If *level* is omitted, then just the current source file and/or subroutine name is printed when either changes. If *level* is 1 or greater, each C source line is printed before it is executed; if *level* is 2 or greater, each assembler statement is also printed. A **v** turns verbose mode off if it is on for any level.

k Kill the program being debugged.

procedure(arg1,arg2,...)
procedure(arg1,arg2,...)/*m*
Execute the named procedure with the given arguments. Arguments can be 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**. This facility is only available if the program was loaded with the –g option.

linenumber **b** *commands*
Set 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 just before the breakpoint and control is returned to *sdb*. Otherwise the *commands* are executed when the breakpoint is encountered and execution continues. Multiple commands are specified by separating them with semicolons. If **k** is used as a command to execute at a breakpoint, control returns to *sdb*, instead of continuing execution.

B Print a list of the currently active breakpoints.

linenumber **d**
Delete 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 Delete all breakpoints.

l Print the last executed line.

linenumber **a**
Announce. If *linenumber* is of the form *proc:number*, the command effectively does a *linenumber* **b l**. If *linenumber* is of the form *proc:*, the command effectively does a *proc:* **b T**.

Miscellaneous commands:

!*command*
The command is interpreted by *sh*(1).

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.

end-of-file character
Scroll. Print 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. When *sdb* is told to display a variable by a command in such a file, the variable name is displayed along with the value. This command may not be nested; < may not appear as a command in a file.

M Print the address maps.

M [?/] [*] *b e f*
Record new values for the address map. The arguments **?** and **/** specify the text and data maps, respectively. The first segment (*b1*, *e1*, *f1*) is changed unless * is specified, in which case the second segment (*b2*, *e2*, *f2*) of the mapping is changed. If fewer than three values are given, the remaining map parameters are left unchanged.

" *string*
Print the given string. The C escape sequences of the form *\character* are recognized, where *character* is a nonnumeric character.

q Exit the debugger.

The following commands also exist and are intended only for debugging the debugger:

V Print the version number.
Q Print a list of procedures and files being debugged.
Y Toggle debug output.

FILES

a.out
core

SEE ALSO

cc(1), f77(1), a.out(4), core(4), syms(4).
sh(1) in the *User's Reference Manual.*

WARNINGS

Breakpoints do not work with older 0410 (non-page-aligned) objects.

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.

BUGS

If a procedure is called when the program is *not* stopped at a breakpoint (such as when a core image is being debugged), all variables are initialized before the procedure is started. This makes it impossible to use a procedure which formats data from a core image.

NAME

size – print section sizes in bytes of common object files

SYNOPSIS

size [**–n**] [**–f**] [**–o**] [**–x**] [**–V**] files

DESCRIPTION

The *size* command produces section size information in bytes for each loaded section in the common object files. The size of the text, data, and bss (uninitialized data) sections is printed, as well as the sum of the sizes of these sections. If an archive file is input to the *size* command the information for all archive members is displayed.

The **-n** option includes NOLOAD sections in the size.

The **-f** option produces full output, that is, it prints the size of every loaded section, followed by the section name in parentheses.

Numbers will be printed in decimal unless either the **–o** or the **–x** option is used, in which case they will be printed in octal or in hexadecimal, respectively.

The **–V** flag will supply the version information on the *size* command.

SEE ALSO

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

WARNING

Since the size of bss sections is not known until link-edit time, the *size* command will not give the true total size of pre-linked objects.

DIAGNOSTICS

size: name: cannot open
if *name* cannot be read.

size: name: bad magic
if *name* is not an appropriate common object file.

NAME

strip – strip symbol and line number information from a common object file

SYNOPSIS

strip [–l] [–x] [–b] [–r] [–V] filename ...

DESCRIPTION

The *strip* command strips the symbol table and line number information from common object files, including archives. Once this has been done, no symbolic debugging access will be available for that file; therefore, this command is normally run only on production modules that have been debugged and tested.

The amount of information stripped from the symbol table can be controlled by using any of the following options:

–l Strip line number information only; do not strip any symbol table information.

–x Do not strip static or external symbol information.

–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.

–V Print the version of the strip command executing on the standard error output.

If there are any relocation entries in the object file and any symbol table information is to be stripped, *strip* will complain and terminate without stripping *filename* unless the **–r** option is used.

If the *strip* command 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 the *ar*(1) command with the **s** option before the archive can be link-edited by the *ld*(1) command. *strip* will produce appropriate warning messages when this situation arises.

strip is used to reduce the file storage overhead taken by the object file.

FILES

TMPDIR/strp* temporary files

TMPDIR is usually /usr/tmp but can be redefined by setting the environment variable **TMPDIR** [see *tempnam*( ) in *tmpnam*(3S)].

SEE ALSO

ar(1), as(1), cc(1), ld(1), tmpnam(3S), a.out(4), ar(4).

DIAGNOSTICS

strip: name: cannot open
if *name* cannot be read.

strip: name: bad magic
if *name* is not an appropriate common 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.

(1)

NAME

tic – terminfo compiler

SYNOPSIS

tic [–v[n]] [–c] file

DESCRIPTION

tic translates a *terminfo*(4) file from the source format into the compiled format. The results are placed in the directory */usr/lib/terminfo*. The compiled format is necessary for use with the library routines described in *curses*(3X).

–vn (verbose) output to standard error trace information showing *tic*'s progress. The optional integer *n* is a number from 1 to 10, inclusive, indicating the desired level of detail of information. If *n* is omitted, the default level is 1. If *n* is specified and greater than 1, the level of detail is increased.

–c only check *file* for errors. Errors in **use**= links are not detected.

file contains one or more *terminfo*(4) terminal descriptions in source format (see *terminfo*(4)). Each description in the file describes the capabilities of a particular terminal. When a **use**=*entry-name* field is discovered in a terminal entry currently being compiled, *tic* reads in the binary from */usr/lib/terminfo* to complete the entry. (Entries created from *file* will be used first. If the environment variable **TERMINFO** is set, that directory is searched instead of */usr/lib/terminfo*.) *tic* duplicates the capabilities in *entry-name* for the current entry, with the exception of those capabilities that explicitly are defined in the current entry.

If the environment variable **TERMINFO** is set, the compiled results are placed there instead of */usr/lib/terminfo*.

FILES

/usr/lib/terminfo/?/* compiled terminal description data base

SEE ALSO

curses(3X), term(4), terminfo(4)
The **curses/terminfo** chapter in the *Programmer's Guide*.

WARNINGS

Total compiled entries cannot exceed 4096 bytes. The name field cannot exceed 128 bytes.

Terminal names exceeding 14 characters will be truncated to 14 characters and a warning message will be printed.

When the **–c** option is used, duplicate terminal names will not be diagnosed; however, when **–c** is not used, they will be.

BUGS

To allow existing executables from the previous release of SYSTEM V/68 to continue to run with the compiled terminfo entries created by the new terminfo compiler, cancelled capabilities will not be marked as cancelled within the terminfo binary unless the entry name has a '+' within it. (Such terminal names are only used for inclusion within other entries via a **use=** entry. Such names would not be used for real terminal names.)

For example:

4415+nl, kf1@, kf2@, ...

4415+base, kf1=\EOc, kf2=\EOd, ...

4415-nl|4415 terminal without keys,
use=4415+nl, use=4415+base,

The above example works as expected; the definitions for the keys do not show up in the *4415–nl* entry. However, if the entry *4415+nl* did not have a plus sign within its name, the cancellations would not be marked within the compiled file and the definitions for the function keys would not be cancelled within *4415–nl*.

DIAGNOSTICS

Most diagnostic messages produced by *tic* during the compilation of the source file are preceded with the approximate line number and the name of the terminal currently being worked on.

mkdir ... returned bad status
The named directory could not be created.

File does not start with terminal names in column one
The first thing seen in the file, after comments, must be the list of terminal names.

Token after a *seek*(2) not NAMES
Somehow the file being compiled changed during the compilation.

Not enough memory for use_list element
or
Out of memory
Not enough free memory was available (*malloc*(3) failed).

Can't open ...
The named file could not be created.

Error in writing ...
The named file could not be written to.

Can't link ... to ...
A link failed.

Error in re-reading compiled file ...
The compiled file could not be read back in.

Premature EOF
The current entry ended prematurely.

Backspaced off beginning of line
This error indicates something wrong happened within *tic*.

Unknown Capability - "..."
The named invalid capability was found within the file.

Wrong type used for capability "..."
For example, a string capability was given a numeric value.

Unknown token type
Tokens must be followed by '@' to cancel, ',' for booleans, '#' for numbers, or '=' for strings.

"...": bad term name
or
Line ...: Illegal terminal name - "..."
Terminal names must start with a letter or digit
The given name was invalid. Names must not contain white space or slashes, and must begin with a letter or digit.

"...": terminal name too long.
An extremely long terminal name was found.

"...": terminal name too short.
A one-letter name was found.

"..." filename too long, truncating to "..."
The given name was truncated to 14 characters due to SYSTEM V/68 file name length limitations.

"..." defined in more than one entry. Entry being used is "...".
An entry was found more than once.

Terminal name "..." synonym for itself
A name was listed twice in the list of synonyms.

At least one synonym should begin with a letter.
At least one of the names of the terminal should begin with a letter.

Illegal character - "..."
The given invalid character was found in the input file.

Newline in middle of terminal name
The trailing comma was probably left off of the list of names.

Missing comma
A comma was missing.

Missing numeric value
The number was missing after a numeric capability.

NULL string value
The proper way to say that a string capability does not exist is to cancel it.

Very long string found. Missing comma?
self-explanatory

Unknown option. Usage is:
An invalid option was entered.

Too many file names. Usage is:
self-explanatory

"..." non-existant or permission denied
The given directory could not be written into.

"..." is not a directory
self-explanatory

"...": Permission denied
access denied.

"...": Not a directory
tic wanted to use the given name as a directory, but it already exists as a file

SYSTEM ERROR!! Fork failed!!!
A *fork*(2) failed.

Error in following up use-links. Either there is a loop in the links or they reference non-existant terminals. The following is a list of the entries involved:
A *terminfo*(4) entry with a **use**=*name* capability either referenced a non-existant terminal called *name* or *name* somehow referred back to the given entry.

NAME

tsort – topological sort

SYNOPSIS

tsort [file]

DESCRIPTION

The *tsort* command 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 understood.

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.

SEE ALSO

lorder(1).

DIAGNOSTICS

Odd data: there is an odd number of fields in the input file.

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 prior to creating the intended new delta. If a directory is named, *unget* behaves as though each file in the directory were specified 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 being taken as the name of an SCCS file to be processed.

Keyletter arguments apply independently to each named file.

–r*SID* Uniquely identifies which delta is no longer intended. (This would have been specified by *get* as the "new delta"). The use of this keyletter is necessary only if two or more outstanding *get*s 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 Causes the retention of the gotten file which would normally be removed from the current directory.

SEE ALSO

delta(1), get(1), sact(1).
help(1) in the *User's Reference Manual*.

DIAGNOSTICS

Use *help*(1) for explanations.

(1)

NAME

val – validate SCCS file

SYNOPSIS

val –
val [–s] [–rSID] [–**m**name] [–ytype] 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 keyletter arguments, which begin with a –, and named files.

val has a special argument, –, which causes reading of 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 keyletter arguments are defined as follows. The effects of any keyletter argument apply independently to each named file on the command line.

–**s** The presence of this argument 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., **r**1 is ambiguous because it physically does not exist but implies 1.1, 1.2, etc., which may exist) or invalid (e. g., **r**1.0 or **r**1.1.0 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.

–mname The argument value *name* is compared with the s-1SCCS %M% keyword in *file*.

–ytype 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 keyletter argument;
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 is returned – a logical **OR** of the codes generated for each command line and file processed.

SEE ALSO

admin(1), delta(1), get(1), prs(1).
help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

val can process up to 50 files on a single command line. Any number above 50 will produce a **core** dump.

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. In the process of performing the copy operation, 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 less 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. An uninterpreted control character may be included 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 and 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.

–cchar 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]
Used to declare keywords. All keywords must be declared.

:asg keyword=value
Used to assign 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
Used to skip 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 solely for the purpose of 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 only occur 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
&
|
```

Parentheses may be used to alter the order of precedence.
Values must be separated from operators or parentheses by at least one blank or tab.

::text

Used for keyword replacement 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

Turn on or off keyword replacement on all lines.

:ctl char

Change 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.

SEE ALSO

ed(1), help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

EXIT CODES

0 – normal
1 – any error

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

```
f.c:
        identification information
f.o:
        identification information
a.out:
        identification information
```

what is intended to be used in conjunction with the command *get*(1), which automatically inserts identifying information, but it can also be used where the information is inserted manually. Only one option exists:

–s Quit after finding the first occurrence of pattern in each file.

SEE ALSO

get(1).
help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Exit status is 0 if any matches are found, otherwise 1. Use *help*(1) for explanations.

BUGS

It is possible that an unintended occurrence of the pattern **@(#)** could be found just by chance, but this causes no harm in nearly all cases.

(1)

NAME

yacc – yet another compiler-compiler

SYNOPSIS

yacc [–vdlt] grammar

DESCRIPTION

The *yacc* command converts a context-free grammar into a set of tables for a simple automaton which executes an LR(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; *lex*(1) is useful for creating lexical analyzers usable by *yacc*.

If the **–v** flag is given, the file **y.output** is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

If the **–d** flag is used, the file **y.tab.h** is generated with the **#define** statements that associate the *yacc*-assigned "token codes" with the user-declared "token names". This allows source files other than **y.tab.c** to access the token codes.

If the **–l** flag is given, the code produced in **y.tab.c** will *not* contain any **#line** constructs. This should only be used after the grammar and the associated actions are fully debugged.

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. However, when *yacc*'s **–t** option is used, this debugging code will be compiled by default. Independent of whether the **–t** option was 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.

FILES

y.output	
y.tab.c	
y.tab.h	defines for token names
yacc.tmp,	
yacc.debug, yacc.acts	temporary files
/usr/lib/yaccpar	parser prototype for C programs

SEE ALSO

lex(1).

The **yacc** chapter of the *Programmer's Guide.*

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 is also reported.

WARNING

Because file names are fixed, at most one *yacc* process can be active in a given directory at a given time.