UNIX
VOL 0.

NAME

inews — submit news articles

SYNOPSIS

inews [**—h**] **—t** *title* [**—n** *newsgroups*] [**—e** *expiration date*]

inews —p [*filename*]

inews —C *newsgroup*

DESCRIPTION

Inews submits news articles to the USENET news network. It is intended as a raw interface, not as a human user interface. Casual users should probably use *postnews* (1) instead.

The first form is for submitting user articles. The body will be read from the standard input. A *title* must be specified as there is no default. Each article belongs to a list of newsgroups. If the **—n** flag is omitted, the list will default to something like *general*. (On ours, it is *general*.) If you wish to submit an article in multiple newsgroups, the *newsgroups* must be separated by commas and/or spaces. If not specified, the expiration date will be set to the local default. The **—f** flag specifies the article's sender. Without this flag, the sender defaults to the user's name. If **—f** is specified, the real sender's name will be included as a Sender line. The **—h** flag specifies that headers are present at the beginning of the article, and these headers should be included with the article header instead of as text. (This mechanism can be used to edit headers and supply additional nondefault headers, but not to specify certain information, such as the sender and article ID, that inews itself generates.)

When posting an article, the environment is checked for information about the sender. If NAME is found, its value is used for the full name, rather than the system value (often in /etc/passwd). This is useful if the system value cannot be set, or when more than one person uses the same login. If ORGANIZATION is found, the value overrides the system default organization. This is useful when a person uses a guest login and is not primarily associated with the organization owning the machine.

The second form is used for receiving articles from other machines. If *filename* is given, the article will be read from the specified file; otherwise the article will be read from the standard input. An expiration date need not be present and a receival date, if present, will be ignored.

After local installation, inews will transmit the article to all systems that subscribe to the newsgroups that the article belongs to.

The third form is for creating new newsgroups. On some systems, this may be limited to specific users such as the super-user or news administrator. (This happens on ours.)

If the file /usr/lib/news/recording is present, it is taken as a list of "recordings" to be shown to users posting news. (This is by analogy to the recording you hear when you dial information in some parts of the country, asking you if you really wanted to do this.) The file contains lines of the form:

newsgroups <tab> filename

for example:

net.all net.recording fa.all fa.recording

Any user posting an article to a newsgroup matching the pattern on the left will be shown the contents of the file on the right. The file is found in the LIB directory (often /usr/lib/news). The user is then told to hit DEL to abort or RETURN to proceed. The intent of this feature is to help companies keep proprietary information from accidently leaking out.

FILES

/usr/spool/news/.sys.nnn temporary articles
/usr/spool/news/*newsgroups*/*article_no*.

	Articles
/usr/spool/oldnews/	Expired articles
/usr/lib/news/active	List of known newsgroups and highest local article numbers in each.
/usr/lib/news/seq	Sequence number of last article
/usr/lib/news/history	List of all articles ever seen
/usr/lib/news/sys	System subscription list

SEE ALSO

Mail(1), binmail(1), getdate(3), msgs(1), news(5), newsrc(5), postnews(1), readnews(1), rec-news(1), sendnews(8), uucp(1), uurec(8),

AUTHORS

Matt Glickman
Mark Horton
Stephen Daniel
Tom R. Truscott

NAME

postnews — submit news articles

SYNOPSIS

postnews [*article*]

DESCRIPTION

Postnews is a shell script that calls *inews*(1) to submit news articles to USENET. It will prompt the user for the title of the article (which should be a phrase suggesting the subject, so that persons reading the news can tell if they are interested in the article) for the newsgroup, and for the distribution.

An omitted newsgroup (from hitting return) will default to *general*.

general is read by everyone on the local machine. Other possible newsgroups include, but are not limited to, *btl.general*, which is read by all users at all Bell Labs sites on USENET, *net.general*, which is read by all users at all sites on USENET, and *net.news*, which is read by users interested in the network news on all sites. There is often a local set of newsgroups, such as *ucb.all*, that circulate within a local set of machines. (In this case, *ucb* newsgroups circulate among machines at the University of California at Berkeley.)

The distribution can be any valid newsgroup name list, and defaults to the same as the newsgroup. (If they are the same, the distribution will be omitted from the headers put into the editor buffer.) A distribution header will, if given, be included in the headers of the article, affecting where the article is distributed to.

After entering the title, newsgroup, and distribution, the user will be placed in an editor. If $EDITOR is set in the environment, that editor will be used. Otherwise, postnews defaults to *vi*(1).

An initial set of headers containing the subject and newsgroups will be placed in the editor, followed by a blank line. The article should be appended to the buffer, after the blank line. These headers can be changed, or additional headers added, while in the editor, if desired.

Optionally, the article will be read from the specified *filename*.

For more sophisticated uses, such as posting news from a program, see *inews*(1).

FILES

SEE ALSO

Mail(1), checknews(1), inews(1), mail(1), readnews(1).

NAME

readnews — read news articles

SYNOPSIS

readnews [—**a** *date*] [—**n** *newsgroups*] [—**t** *titles*] [—**lprxhfuM**] [—**c** [*mailer*]]

readnews —**s**

DESCRIPTION

readnews without argument prints unread articles. There are several interfaces available:

Flag	Interface
default	A *msgs*(1) like interface.
—**M**	An interface to *Mail*(1).
—**c**	A */bin/mail*(1)—like interface.

—**c** *"mailer"*
All selected articles written to a temporary file. Then the mailer is invoked. The name of the temporary file is referenced with a "%". Thus, "mail —f %" will invoke mail on a temporary file consisting of all selected messages.

—**p** All selected articles are sent to the standard output. No questions asked.

—**l** Only the titles output. The *.newsrc* file will not be updated.

The —**r** flag causes the articles to be printed in reverse order. The —**f** flag prevents any followup articles from being printed. The —**h** flag causes articles to be printed in a less verbose format, and is intended for terminals running at 300 baud. the —**u** flag causes the **.newsrc** file to be updated every 5 minutes, in case of an unreliable system. (Note that if the newsrc file is updated, the **x** command will not restore it to its original contents.)

The following flags determine the selection of articles.

—**n** *newsgroups*
Select all articles that belong to *newsgroups*.

—**t** *titles* Select all articles whose titles contain one of the strings specified by *titles*.

—**a** [**date**]
Select all articles that were posted past the given *date* (in *getdate*(3) format).

—x Ignore *.newsrc* file. That is, select articles that have already been read as well as new ones.

readnews maintains a *.newsrc* file in the user's home directory that specifies all news articles already read. It is updated at the end of each reading session in which the —**x** or —**l** options weren't specified. If the environment variable NEWSRC is present, it should be the path name of a file to be used in place of .newsrc.

If the user wishes, an options line may be placed in the *.newsrc* file. This line starts with the word **options** (left justified) followed by the list of standard options just as they would be typed on the command line. Such a list may include: the —**n** flag along with a newsgroup list; a favorite interface; and/or the —**r** or —**t** flag. Continuation lines are specified by following lines beginning with a space or tab character. Similarly, options can be specified in the **NEWSOPTS** environment parameter. Where conflicts exist, option on the command line take precedence, followed by the *.newsrc* **options** line, and lastly the **NEWSOPTS** parameter.

readnews —**s** will print the newsgroup subscription list.

When the user uses the reply command of the *msgs*(1) or */bin/mail*(1) interfaces, the environment parameter **MAILER** will be used to determine which mailer to use. The default is usually /bin/mail.

If the user so desires, he may specify a specific paging progam for articles. The environment parameter **PAGER** should be set to the paging program. The name of the article is referenced with a '%', as in the —c option. If no '%' is present, the article will be piped to the program. Paging may be disabled by setting **PAGER** to a null value.

COMMANDS

This section lists the commands you can type to the msgs and /bin/mail interface prompts. The msgs interface will suggest some common commands in brackets. Just hitting return is the same as typing the first command. For example, "[ynq]" means that the commands "y" (yes), "n" (no), and "q" (quit) are common responses, and that "y" is the default.

Command — Meaning

y — Yes. Prints current article and goes on to next.

n — No. Goes on to next article without printing current one. In the /bin/mail interface, this means "go on to the next article", which will have the same effect as "y" or just hitting return.

q — Quit. The .newsrc file will be updated if —l or —x were not on the command line.

c — Cancel the article. Only the author or the super user can do this.

r — Reply. Reply to article's author via mail. You are placed in your EDITOR with a header specifying To, Subject, and References lines taken from the message. You may change or add headers, as appropriate. You add the text of the reply after the blank line, and then exit the editor. The resulting message is mailed to the author of the article.

rd — Reply directly. You are placed in $MAILER ("mail" by default) in reply to the author. Type the text of the reply and then control-D.

f [title] — Submit a follow up article. Normally you should leave off the title, since the system will generate one for you. You will be placed in your EDITOR to compose the text of the followup.

fd — Followup directly, without edited headers. This is like *f*, but the headers of the article are not included in the editor buffer.

N [newsgroup] — Go to the next newsgroup or named newsgroup.

s [file] — Save. The article is appended to the named file. The default is "Articles". If the first character of the file name is '|', the rest of the file name is taken as the name of a program, which is executed with the text of the article as standard input. If the first character of the file name is '/', it is taken as a full path name of a file. If $NEWSBOX (in the environment) is set to a full path name, and the file contains no '/', the file is saved in $NEWSBOX. Otherwise, it is saved relative to $HOME.

— Report the name and size of the newsgroup.

e — Erase. Forget that this article was read.

h — Print a more verbose header.

H — Print a very verbose header, containing all known information about the article.

U — Unsubscribe from this newsgroup. Also goes on to the next newsgroup.

d — Read a digest. Breaks up a digest into separate articles and permits you to read and reply to each piece.

D — Decrypt. Invokes a Caesar decoding program on the body of the message. This is used to decrypt rotated jokes posted to net.jokes. Such jokes are usually obscene or otherwise offensive to some groups of people, and so are rotated to avoid accidental

decryption by people who would be offended. The title of the joke should indicate the nature of the problem, enabling people to decide whether to decrypt it or not.

Normally the Caesar program does a character frequency count on each line of the article separately, so that lines which are not rotated will be shown in plain text. This works well unless the line is short, in which case it sometimes gets the wrong rotation. An explicit *number* rotation (usually 13) may be given to force a particular shift.

v Print the current version of the news software.

! Shell escape.

number
Go to number.

+[n] Skip n articles. The articles skipped are recorded as "unread" and will be offered to you again the next time you read news.

— Go back to last article. This is a toggle, typing it twice returns you to the original article.

x Exit. Like quit except that .newsrc is not updated.

X system
Transmit article to the named system.

The commands c, f, fd, r, rd, e, h, H, and s can be followed by —'s to refer to the previous article. Thus, when replying to an article using the msgs interface, you should normally type "r—" (or "re-") since by the time you enter a command, you are being offerred the next article.

EXAMPLES

readnews
Read all unread articles using the *msgs*(1) interface. The *.newsrc* file is updated at the end of the session.

readnews —c "ed %" —l
Invoke the *ed*(1) text editor on a file containing the titles of all unread articles. The *.newsrc* file is **not** updated at the end of the session.

readnews —n all !fa.all —M —r
Read all unread articles except articles whose newsgroups begin with "fa." via *Mail*(1) in reverse order. The *.newsrc* file is updated at the end of the session.

readnews —p —n all —a last thursday
Print every unread article since last Thursday. The *.newsrc* file is updated at the end of the session.

readnews —p > /dev/null &
Discard all unread news. This is useful after returning from a long trip.

FILES

/usr/spool/news/*newsgroup/number*	News articles
/usr/lib/news/active	Active newsgroups and numbers of articles
/usr/lib/news/help	Help file for *msgs*(1) interface
~/.newsrc	Options and list of previously read articles

SEE ALSO

checknews(1), inews(1), sendnews(8), recnews(8), uurec(8), msgs(1), Mail(1), mail(1), news(5), newsrc(5)

postnews(1)

AUTHORS

Matt Glickman
Mark Horton
Stephen Daniel
Tom R. Truscott

NAME

news — USENET network news article, utility files

DESCRIPTION

There are two formats of news articles: **A** and **B.** **A** format is the only format that version 1 netnews systems can read or write. Systems running the version 2 netnews can read either format and there are provisions for the version 2 netnews to write in **A** format. **A** format looks like this:

A*article-ID*
newsgroups
path
date
title
Body of article

Only version 2 netnews systems can read and write **B** format. **B** format contains two extra pieces of information: receival date and expiration date. The basic structure of a **B** format file consists of a series of headers and then the body. A header field is defined as a line with a capital letter in the 1st column and a colon somewhere on the line. Unrecognized header fields are ignored. News is stored in the same format transmitted, see "Standard for the Interchange of USENET Messages" for a full description. The following fields are among those recognized:

Header	Information
From:	*user@host.domain[.domain ...] (Full Name)*
Newsgroups:	*Newsgroups*
Message-ID:	*<Unique Identifier>*
Subject:	*descriptive title*
Date:	*Date Posted*
Date-Received:	*Date received on local machine*
Expires:	*Expiration Date*
Reply-To:	*Address for mail replies*
References:	*Article ID of article this is*
Control:	*Text of a control message*

Here is an example of an article:

Relay-Version: B 2.10 2/13/83 cbosgd.UUCP
Posting-Version: B 2.10 2/13/83 eagle.UUCP
Path: cbosgd!mhuxj!mhuxt!eagle!jerry
From: jerry@eagle.uucp (Jerry Schwarz)
Newsgroups: net.general
Subject: Usenet Etiquette -- Please Read
Message-ID: <642@eagle.UUCP>
Date: Friday, 19-Nov-82 16:14:55 EST
Followup-To: net.news
Expires: Saturday, 1-Jan-83 00:00:00 EST
Date-Received: Friday, 19-Nov-82 16:59:30 EST
Organization: Bell Labs, Murray Hill

The body of the article comes here, after a blank line.

A *sys* file line has four fields, each seperated by colons:

system-name:subscriptions:flags:transmission command

Of these fields, only the *system-name* and *subscriptions* need to be present.

The *system name* is the name of the system being sent to. The *subscriptions* is the list of newsgroups to be transmitted to the system. The *flags* are a set of letters describing how the article should be transmitted. The default is B. Valid flags include A (send in A format), B (send in B format), N (use ihave/sendme protocol), U (use uux -c and the name of the stored article in a %s string).

The *transmission command* is executed by the shell with the article to be transmitted as the standard input. The default is **uux — —z —r** *sysname*!**rnews.** Some examples:

xyz:net.all
oldsys:net.all,fa.all,to.oldsys:A
berksys:net.all,ucb.all::/usr/lib/news/sendnews —b berksysrnews
arpasys:net.all,arpa.all::/usr/lib/news/sendnews —a rnews@arpasys
old2:net.all,fa.all:A:/usr/lib/sendnews —o old2rnews
user:fa.sf-lovers::mail user

Somewhere in a *sys* file, there must be a line for the host system. This line has no *flags* or *commands*. A # as the first character in a line denotes a comment.

The history, active, and ngfile files have one line per item.

SEE ALSO

inews(1), postnews(1), sendnews(8), uurec(8), readnews(1)

NAME

newsrc — information file for readnews(1) and checknews(1)

DESCRIPTION

The *.newsrc* file contains the list of previously read articles and an optional options line for *readnews(1)* and *checknews(1)*. Each newsgroup that articles have been read from has a line of the form:

newsgroup: range

The *range* is a list of the articles read. It is basically a list of no.'s separated by commas with sequential no.'s collapsed with hyphens. For instance:

general: 1-78,80,85-90
fa.info-cpm: 1-7
net.news: 1
fa.info-vax! 1-5

If the : is replaced with an ! (as in info-vax above) the newsgroup is not subscribed to and will not be shown to the user.

An options line starts with the word **options** (left-justified). Then there are the list of options just as they would be on the command line. For instance:

options —n all !fa.sf-lovers !fa.human-nets —r
options —c —r

A string of lines beginning with a space or tab after the initial options line will be considered continuation lines.

FILES

~/.newsrc options and list of previously read articles

SEE ALSO

readnews(1), checknews(1)

NAME

recnews — receive unprocessed articles via mail

SYNOPSIS

/usr/lib/news/recnews [*newsgroup* [*sender*]]

DESCRIPTION

Recnews reads a letter from the standard input; determines the article title, sender, and newsgroup; and gives the body to inews with the right arguments for insertion.

If *newsgroup* is omitted, the to line of the letter will be used. If *sender* is omitted, the sender will be determined from the from line of the letter. The title is determined from the subject line.

SEE ALSO

inews(1), uurec(8), sendnews(8), readnews(1), checknews(1)

NAME

sendnews — send news articles via mail

SYNOPSIS

sendnews [**—o**] [**—a**] [**—b**] [**—n** newsgroups] destination

DESCRIPTION

sendnews reads an article from it's standard input, performs a set of changes to it, and gives it to the mail program to mail it to *destination.*

An 'N' is prepended to each line for decoding by *uurec(1).*

The **—o** flag handles old format articles.

The **—a** flag is used for sending articles via the **ARPANET.** It maps the article's path from *uucphost!xxx* to *xxx@arpahost.*

The **—b** flag is used for sending articles via the **Berknet.** It maps the article's path from *uucphost!xxx* to *berkhost:xxx.*

The **—n** flag changes the article's newsgroup to the specified *newsgroup.*

SEE ALSO

inews(1), uurec(8), recnews(8), readnews(1), checknews(1)

4.2 Version

User Contributed Software

Note: This "User Contributed Software" is part of the standard Berkeley 4.2BSD release. It is included as part of MORE/bsd for the use of our customers, but it is not supported in any way by MT XINU. NO WARRANTY, EXPRESS OR IMPLIED, OF ANY KIND, IS MADE REGARDING THIS SOFTWARE.

NAME

ci — check in RCS revisions

SYNOPSIS

ci [options] file ...

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

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

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

If *rev* is omitted, *ci* derives the new revision number from the caller's last lock. If the caller has locked the tip revision of a branch, the new revision is appended to that branch. The new revision number is obtained by incrementing the tip revision number. If the caller locked a non-tip revision, a new branch is started at that revision by incrementing the highest branch number at that revision. The default initial branch and level numbers are 1. If the caller holds no lock, but he

is the owner of the file and locking is not set to *strict*, then the revision is appended to the trunk.

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

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

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

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

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

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

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

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

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

—n*name* assigns the symbolic name *name* to the number of the checked-in revision. *Ci* prints an error message if *name* is already assigned to another number.

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

—s*state* sets the state of the checked-in revision to the identifier *state*. The default is *Exp*.

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

DIAGNOSTICS

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

FILE MODES

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

FILES

The caller of the command must have read/write permission for the directories containing the RCS file and the working file, and read permission for the RCS file itself. A number of temporary files are created. A semaphore file is created in the directory containing the RCS file. *Ci* always creates a new RCS file and unlinks the old one. This strategy makes links to RCS files useless.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.1 ; Release Date: 83/04/04 .

SEE ALSO

co (1), ident(1), rcs (1), rcsdiff (1), rcsintro (1), rcsmerge (1), rlog (1), rcsfile (5), sccstorcs (8).

Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

NAME

co — check out RCS revisions

SYNOPSIS

co [options] file ...

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

KEYWORD SUBSTITUTION

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

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

Keywords and their corresponding values:

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

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

$Header$ A standard header containing the RCS file name, the revision number, the

date, the author, and the state.

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

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

$Revision$ The revision number assigned to the revision.

$Source$ The full pathname of the RCS file.

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

DIAGNOSTICS

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

EXAMPLES

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

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

FILE MODES

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

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

FILES

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

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

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.1 ; Release Date: 83/04/04 .

SEE ALSO

ci (1), ident(1), rcs (1), rcsdiff (1), rcsintro (1), rcsmerge (1), rlog (1), rcsfile (5), sccstorcs (8).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

LIMITATIONS

The option -d gets confused in some circumstances, and accepts no date before 1970. There is no way to suppress the expansion of keywords, except by writing them differently. In

nroff and troff, this is done by embedding the null-character '\&' into the keyword.

BUGS

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

NAME

comp — compose a message

SYNOPSIS

comp [—editor editor] [—form formfile] [file] [—use] [—nouse] [—help]

DESCRIPTION

Comp is used to create a new message to be mailed. If *file* is not specified, the file named "draft" in the user's MH directory will be used. *Comp* copies a message form to the file being composed and then invokes an editor on the file. The default editor is /bin/ned, which may be overridden with the '—editor' switch or with a profile entry "Editor:". The default message form contains the following elements:

```
To:
cc:
Subject:
--------
```

If the file named "components" exists in the user's MH directory, it will be used instead of this form. If '—form formfile' is specified, the specified formfile (from the MH directory) will be used as the skeleton. The line of dashes or a blank line must be left between the header and the body of the message for the message to be identified properly when it is sent (see *send*(1)). The switch '—use' directs *comp* to continue editing an already started message. That is, if a *comp* (or *dist*, *repl*, or *forw*) is terminated without sending the message, the message can be edited again via "comp —use".

If the specified file (or draft) already exists, *comp* will ask if you want to delete it before continuing. A reply of **No** will abort the *comp*, **yes** will replace the existing draft with a blank skeleton, **list** will display the draft, and **use** will use it for further composition.

Upon exiting from the editor, *comp* will ask "What now?". The valid responses are **list**, to list the draft on the terminal; **quit**, to terminate the session and preserve the draft; **quit delete**, to terminate, then delete the draft; **send**, to send the message; **send verbose**, to cause the delivery process to be monitored; **edit <editor>**, to invoke <editor> for further editing; and **edit**, to re-edit using the same editor that was used on the preceding round unless a profile entry "<lasteditor>—next: <editor>" names an alternative editor.

Comp does not affect either the current folder or the current message.

FILES

/etc/mh/components	The message skeleton
or <mh-dir>/components	Rather than the standard skeleton
$HOME/mh_profile	The user profile
<mh-dir>/draft	The default message file
/usr/new/send	To send the composed message

PROFILE COMPONENTS

Path:	To determine the user's MH directory
Editor:	To override the use of /bin/ned as the default editor
<lasteditor>—next:	editor to be used after exit from <lasteditor>

DEFAULTS

'file' defaults to draft
'—editor' defaults to /bin/ned
'—nouse'

NAME

dist — redistribute a message to additional addresses

SYNOPSIS

dist [+folder] [msg] [—form formfile] [—editor editor] [—annotate] [—noannotate] [—inplace] [—noinplace] [—help]

DESCRIPTION

Dist is similar to *forw*. It prepares the specified message for redistribution to addresses that (presumably) are not on the original address list. The file "distcomps" in the user's MH directory, or a standard form, or the file specified by '—form formfile' will be used as the blank components file to be prepended to the message being distributed. The standard form has the components "Distribute-to:" and "Distribute-cc:". When the message is sent, "Distribution-Date: date", "Distribution-From: name", and "Distribution-Id: id" (if '—msgid' is specified to *send*;) will be prepended to the outgoing message. Only those addresses in "Distribute-To", "Distribute-cc", and "Distribute-Bcc" will be sent. Also, a "Distribute-Fcc: folder" will be honored (see *send*;).

Send recognizes a message as a redistribution message by the existence of the field "Distribute-To:", so don't try to redistribute a message with only a "Distribute-cc:".

If the '—annotate' switch is given, each message being distributed will be annotated with the lines:

Distributed: date
Distributed: Distribute-to: names

where each "to" list contains as many lines as required. This annotation will be done only if the message is sent directly from *dist*. If the message is not sent immediately from *dist* (i.e., if it is sent later via *send*;), "comp —use" may be used to re-edit and send the constructed message, but the annotations won't take place. The '—inplace' switch causes annotation to be done in place in order to preserve links to the annotated message.

See *comp* for a description of the '—editor' switch and for options upon exiting from the editor.

If a +folder is specified, it will become the current folder, and the current message will be set to the message being redistributed.

FILES

/etc/mh/components	The message skeleton
or <mh-dir>/components	Rather than the standard skeleton
$HOME/mh_profile	The user profile
<mh-dir>/draft	The default message file
/usr/bin/send	To send the composed message

PROFILE COMPONENTS

Path:	To determine the user's MH directory
Editor:	To override the use of /bin/ned as the default editor
<lasteditor>—next:	editor to be used after exit from <lasteditor>

DEFAULTS

'+folder' defaults to the current folder
'msg' defaults to cur
'—editor' defaults to /bin/ned
'—noannotate'
'—noinplace'

NAME

file — file message(s) in (an)other folder(s)

SYNOPSIS

file [—src +folder] [msgs] [—link] [—preserve] +folder ... [—nolink] [—nopreserve] [—file file] [—nofile] [—help]

DESCRIPTION

File moves (*mv*(1)) or links (*ln*(1)) messages from a source folder into one or more destination folders. If you think of a message as a sheet of paper, this operation is not unlike filing the sheet of paper (or copies) in file cabinet folders. When a message is filed, it is linked into the destination folder(s) if possible, and is copied otherwise. As long as the destination folders are all on the same file system, multiple filing causes little storage overhead. This facility provides a good way to cross-file or multiply-index messages. For example, if a message is received from Jones about the ARPA Map Project, the command

file cur +jones +Map

would allow the message to be found in either of the two folders 'jones' or 'Map'.

The option '—file file' directs *file* to use the specified file as the source message to be filed, rather than a message from a folder.

If a destination folder doesn't exist, *file* will ask if you want to create one. A negative response will abort the file operation.

'—link' preserves the source folder copy of the message (i.e., it does a *ln*(1) rather than a *mv*(1)), whereas, '—nolink' deletes the "filed" messages from the source folder. Normally, when a message is filed, it is assigned the next highest number available in each of the destination folders. Use of the '—preserve' switch will override this message "renaming", but name conflicts may occur, so use this switch cautiously. (See *pick* for more details on message numbering.)

If '—link' is not specified (or '—nolink' is specified), the filed messages will be removed (*unlink*(2)) from the source folder.

If '—src +folder' is given, it will become the current folder for future MH commands. If neither '—link' nor 'all' are specified, the current message in the source folder will be set to the last message specified; otherwise, the current message won't be changed.

FILES

$HOME/mh_profile The user profile

PROFILE COMPONENTS

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder
Folder—Protect:	To set mode when creating a new folder

DEFAULTS

'—src +folder' defaults to the current folder
'msgs' defaults to cur
'—nolink'
'—nopreserve'
'—nofile'

CONTEXT

If '—src +folder' is given, it will become the current folder for future MH commands. If neither '—link' nor 'all' are specified, the current message in the source folder will be set to the last message specified; otherwise, the current message won't be changed.

NAME

folder — set/list current folder/message

SYNOPSIS

folder [+folder] [msg] [—all] [—fast] [—nofast] [—up] [—down] [—header] [—noheader] [—total] [—nototal] [—pack] [—nopack] [—help]

folders <equivalent to 'folder —all'>

DESCRIPTION

Since the MH environment is the shell, it is easy to lose track of the current folder from day to day. ***Folder*** will list the current folder, the number of messages in it, the range of the messages (low-high), and the current message within the folder, and will flag a selection list or extra files if they exist. An example of the output is:

inbox+ has 16 messages (3— 22); cur= 5.

If a '+folder' and/or 'msg' are specified, they will become the current folder and/or message. An '—all' switch will produce a line for each folder in the user's MH directory, sorted alphabetically. These folders are preceded by the read-only folders, which occur as mh_profile "cur—" entries. For example,

```
                Folder      # of messages (  range  ); cur msg (other files)
    /fsd/rs/m/tacc  has   35 messages (  1— 35); cur= 23.
/rnd/phyl/Mail/EP  has   82 messages (  1—108); cur= 82.
                ff  has    4 messages (  1—  4); cur=  1.
            inbox+  has   16 messages (  3— 22); cur=  5.
                mh  has   76 messages (  1— 76); cur= 70.
             notes  has    2 messages (  1—  2); cur=  1.
              ucom  has  124 messages (  1—124); cur=  6; (select).

                TOTAL= 339 messages in   7 Folders.
```

The "+" after inbox indicates that it is the current folder. The "(select)" indicates that the folder ucom has a selection list produced by *pick*. If "others" had appeared in parentheses at the right of a line, it would indicate that there are files in the folder directory that don't belong under the MH file naming scheme.

The header is output if either an '—all' or a '—header' switch is specified; it is suppressed by '—noheader'. Also, if *folder* is invoked by a name ending with "s" (e.g., *folders*), '—all' is assumed. A '—total' switch will produce only the summary line.

If '—fast' is given, only the folder name (or names in the case of '—all') will be listed. (This is faster because the folders need not be read.)

The switches '—up' and '—down' change the folder to be the one above or below the current folder. That is, "folder —down" will set the folder to "<current—folder>/select", and if the current folder is a selection-list folder, "folder —up" will set the current folder to the parent of the selection-list. (See *pick* for details on selection-lists.)

The '—pack' switch will compress the message names in a folder, removing holes in message numbering.

FILES

$HOME/mh_profile	The user profile
/bin/ls	To fast-list the folders

PROFILE COMPONENTS

Path:	To determine the user's MH directory

Current-Folder: To find the default current folder

DEFAULTS

'+folder' defaults to the current folder
'msg' defaults to none
'—nofast'
'—noheader'
'—nototal'
'—nopack'

CONTEXT

If '+folder' and/or 'msg' are given, they will become the current folder and/or message.

NAME

forw — forward messages

SYNOPSIS

forw [+folder] [msgs] [—editor editor] [—form formfile] [—annotate] [—noannotate] [—inplace] [—noinplace] [—help]

DESCRIPTION

Forw may be used to prepare a message containing other messages. It constructs the new message from the components file or '—form formfile' (see *comp*(1)), with a body composed of the message(s) to be forwarded. An editor is invoked as in *comp*, and after editing is complete, the user is prompted before the message is sent.

If the '—annotate' switch is given, each message being forwarded will be annotated with the lines

Forwarded: date
Forwarded: To: names
Forwarded: cc: names

where each "To:" and "cc:" list contains as many lines as required. This annotation will be done only if the message is sent directly from *forw*. If the message is not sent immediately from *forw*, "comp —use" may be used in a later session to re-edit and send the constructed message, but the annotations won't take place. The '—inplace' switch permits annotating a message in place in order to preserve its links.

See *comp* for a description of the '—editor' switch.

FILES

/etc/mh/components	The message skeleton
or <mh-dir>/components	Rather than the standard skeleton
$HOME/mh_profile	The user profile
<mh-dir>/draft	The default message file
/usr/bin/send	To send the composed message

PROFILE COMPONENTS

Path:	To determine the user's MH directory
Editor:	To override the use of /bin/ned as the default editor
Current-Folder:	To find the default current folder
<lasteditor>—next:	editor to be used after exit from <lasteditor>

DEFAULTS

'+folder' defaults to the current folder
'msgs' defaults to cur
'—editor' defaults to /bin/ned
'—noannotate'
'—noinplace'

CONTEXT

If a +folder is specified, it will become the current folder, and the current message will be set to the first message being forwarded.

NAME

iconc — compile and link Icon programs

SYNOPSIS

iconc [option ...] file ...

DESCRIPTION

Iconc is a compiler for Version 5 of the Icon programming language. Compilation consists of four phases: *translation*, *linking*, *assembling*, and *loading*. During translation, each Icon source file is translated into an intermediate language; during linking, the intermediate language files are combined and a single assembly code output file is produced which is then assembled; during loading, the assembled program is loaded with the Icon runtime system libraries, producing an executable file. Unless the **–o** option is specified, the name of the resulting executable file is formed by deleting the suffix of the first file named on the command line.

Files whose names end in '.icn' are assumed to be Icon source programs; they are translated, and the intermediate code is left in two files of the same name with '.u1' and '.u2' substituted for '.icn'. The intermediate code files are normally deleted when compilation has finished. Files whose names end in '.u1' or '.u2' are assumed to be intermediate code files from a previous translation (only one should be named — the other is assumed); these files are included in the linking phase after any '.icn' files have been translated. Files whose names end in '.c' or '.o' are assumed to be external functions. Any '.c' file is compiled using *cc* (1) to produce a '.o' file. A '.u1', '.u2', '.c', or '.o' file that is explicitly named is not deleted.

The following options are recognized by *iconc*.

–c Suppress the linking and loading phases. The intermediate code files are not deleted.

–m
Preprocess each '.icn' source file with the *m4* (1) macro processor before translation.

–o *output*
Name the executable file *output*.

–s Suppress any informative messages from the translator and linker. Normally, both informative messages and error messages are sent to standard error output.

–t Arrange for **&trace** to have an initial value of –1 when the program is executed. Normally, **&trace** has an initial value of 0.

–u Issue warning messages for undeclared identifiers in the program. The warnings are issued during the linking phase.

When an Icon program is executed, a number of environment variables are examined to determine certain execution parameters. The values assigned to these variables should be numbers. The variables that affect execution and the interpretations of their values are as follows:

TRACE
Initialize the value of **&trace**. If this variable has a value, it overrides the translation-time **–t** option.

NBUFS
The number of i/o buffers to use for files. When a file is opened, it is assigned an i/o buffer if one is available and the file is not a tty. If no buffer is available, the file is not buffered. **&input**, **&output**, and **&errout** are buffered if buffers are available. On VAX systems, ten buffers are allocated initially; on PDP-11 systems, five buffers are allocated initially.

NOERRBUF

If set, **&errout** is not buffered.

STRSIZE
The initial size of the string space, in bytes. The string space grows if necessary, but it never shrinks. On VAX systems, the string space is initially 51,200 bytes; on PDP-11 systems, 10,240 bytes initially.

HEAPSIZE
The initial size of the heap, in bytes. The heap grows if necessary, but it never shrinks. On VAX systems, the heap is initially 51,200 bytes; on PDP-11 systems, 10,240 bytes initially.

NSTACKS
The number of stacks initially available for co-expressions. On VAX systems, four stacks are initially allocated; on PDP-11 systems, two stacks are initially allocated. More are automatically allocated if needed. It is unwise to set *NSTACKS* to 1.

STKSIZE
The size of each co-expression stack, in words. On VAX systems, stacks are normally 2000 words; on PDP-11 systems, stacks are normally 1000 words.

PROFILE
Turn on execution profiling of the runtime system. The value of this variable specifies the sampling resolution, in words. If the value is zero, profiling is not done. When a profiled program finishes, a file named 'mon.out' is created containing the results of the profile. The program *prof*(1) can be used to examine the results. This produces a profile of the runtime system, not the user program.

FILES

mon.out	results of profiling
v5g/cmp/bin/utran	icon translator
v5g/cmp/bin/ulink	icon linker
v5g/cmp/bin/libi.a	icon runtime library

SEE ALSO

The Icon Programming Language, Ralph E. Griswold and Madge T. Griswold, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1983.

Installation and Maintenance Guide for Release 5g of Icon, Department of Computer Science, The University of Arizona, March 1983.

cc(1), icont(1), ld(1), m4(1), prof(1), monitor(3)

BUGS

Because of the way that co-expressions are implemented, there is a possibility that programs in which they are used may malfunction mysteriously.

Integer overflow on multiplication is not detected.

If the **-m** option is used, line numbers reported in error messages or tracing messages are from the file after, not before, preprocessing.

NAME

icont — translate Icon programs for interpretive execution

SYNOPSIS

icont [option ...] file ... [**−x** arg ...]

DESCRIPTION

Icont is a translator for Version 5 of the Icon programming language, which produces a file suitable for interpretation by the Icon interpreter. Translation consists of two phases: *translation* and *linking*. During translation, each Icon source file is translated into an intermediate language; during linking, the intermediate language files are combined and a single output file is produced. The output file from the linker is referred to as an *interpretable* file. Unless the **−o** option is specified, the name of the resulting interpretable file is formed by deleting the suffix of the first input file named on the command line. If the **−x** argument is used, the file is automatically executed by the interpreter and any arguments following the **−x** are passed as execution arguments to the Icon program itself.

Files whose names end in '.icn' are assumed to be Icon source programs; they are translated, and the intermediate code is left in two files of the same name with '.u1' and '.u2' substituted for '.icn'. The intermediate code files normally are deleted when compilation has finished. Files whose names end in '.u1' or '.u2' are assumed to be intermediate code files from a previous translation (only one should be named — the other is assumed); these files are included in the linking phase after any '.icn' files have been translated. A '.u1' or '.u2' file that is explicitly named is not deleted. Icon source programs may be read from standard input. The argument **−** signifies the use of standard input as a source file. In this case, the intermediate code is placed in 'stdin.u1' and 'stdin.u2' and the interpretable file is 'stdin'.

The following options are recognized by *icont*.

−c Suppress the linking phase. The intermediate code files are not deleted.

−m
Preprocess each '.icn' source file with the *m4* (1) macro processor before translation.

−o *output*
Name the interpretable file *output*.

−s Suppress any informative messages from the translator and linker. Normally, both informative messages and error messages are sent to standard error output.

−t Arrange for **&trace** to have an initial value of −1 when the program is executed. Normally, **&trace** has an initial value of 0.

−u Issue warning messages for undeclared identifiers in the program. The warnings are issued during the linking phase.

The interpretable file produced by the Icon linker is *directly executable*. For example, the command

icont hello.icn

produces a file named **hello** that can be run by the command

hello

The method used to make interpretable files appear to be directly executable is system dependent. See the Icon installation guide for complete details. For most intents and purposes, interpretable files are executable programs in the same sense that files produced by *ld* (1) are executable programs.

Arguments can be passed to the Icon program by following the program name with the arguments. Any such arguments are passed to the main procedure as a list of strings.

When an Icon program is executed, a number of environment variables are examined to determine certain execution parameters. The values assigned to these variables should be numbers. The variables that affect execution and the interpretations of their values are as follows:

TRACE

Initialize the value of **&trace**. If this variable has a value, it overrides the translation-time **–t** option.

NBUFS

The number of i/o buffers to use for files. When a file is opened, it is assigned an i/o buffer if one is available and the file is not a tty. If no buffer is available, the file is not buffered. **&input**, **&output**, and **&errout** are buffered if buffers are available. On VAX systems, ten buffers are allocated initially; on PDP-11 systems, five buffers are allocated initially.

NOERRBUF

If set, **&errout** is not buffered.

STRSIZE

The initial size of the string space, in bytes. The string space grows if necessary, but it never shrinks. On VAX systems, the string space is initially 51,200 bytes; on PDP-11 systems, 10,240 bytes initially.

HEAPSIZE

The initial size of the heap, in bytes. The heap grows if necessary, but it never shrinks. On VAX systems, the heap is initially 51,200 bytes; on PDP-11 systems, 10,240 bytes initially.

NSTACKS

The number of stacks initially available for co-expressions. On VAX systems, four stacks are initially allocated; on PDP-11 systems, two stacks are initially allocated. More are automatically allocated if needed. It is unwise to set *NSTACKS* to 1.

STKSIZE

The size of each co-expression stack, in words. On VAX systems, stacks are normally 2000 words; on PDP-11 systems, stacks are normally 1000 words.

PROFILE

Turn on execution profiling of the runtime system. The value of this variable specifies the sampling resolution, in words. If the value is zero, profiling is not done. When a profiled program finishes, a file named 'mon.out' is created containing the results of the profile. The program *prof* (1) can be used to examine the results. This produces a profile of the runtime system, not the user program.

FILES

v5g/int/bin/utran	icon translator
v5g/int/bin/ulink	icon linker
v5g/int/bin/iconx	icon interpreter
mon.out	results of profiling

SEE ALSO

The Icon Programming Language, Ralph E. Griswold and Madge T. Griswold, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1983.

Installation and Maintenance Guide for Release 5g of Icon, Department of Computer Science, The University of Arizona, March 1983.

iconc(1), m4(1), prof(1), monitor(3)

BUGS

Downward compatibility of interpretable files will not be maintained in subsequent releases of Icon. No checks are performed to determine if the interpretable file and the interpreter are compatible. Peculiar program behavior is the only indication of such incompatibility.

Interpretable files do not stand alone; the Icon interpreter must be present on the system. This implies that an interpretable file produced on one system will not work on another system unless the Icon interpreter is in the same place on both systems and that the interpreter is of the same version of Icon as the translator that produced the interpretable file.

Because of the way that co-expressions are implemented, there is a possibility that programs in which they are used may malfunction mysteriously.

Integer overflow on multiplication is not detected.

If the **-m** option is used, line numbers reported in error messages or tracing messages are from the file after, not before, preprocessing.

NAME

ident — identify files

SYNOPSIS

ident file ...

DESCRIPTION

Ident searches the named files for all occurrences of the pattern *$keyword:...$*, where *keyword* is one of

Author
Date
Header
Locker
Log
Revision
Source
State

These patterns are normally inserted automatically by the RCS command *co (1)*, but can also be inserted manually.

Ident works on text files as well as object files. For example, if the C program in file f.c contains

```
char rcsid[] = "$Header:  Header information $";
```

and f.c is compiled into f.o, then the command

```
ident  f.c  f.o
```

will print

```
f.c:
     $Header:  Header information $
f.o:
     $Header:  Header information $
```

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.0 ; Release Date: 82/12/04 .

SEE ALSO

ci (1), co (1), rcs (1), rcsdiff(1), rcsintro (1), rcsmerge (1), rlog (1), rcsfile (5).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

NAME

inc — incorporate new mail

SYNOPSIS

inc [+folder] [—audit audit-file] [—help]

DESCRIPTION

Inc incorporates mail from the user's incoming mail drop (mail) into an MH folder. If '+folder' isn't specified, the folder named "inbox" in the user's MH directory will be used. The new messages being incorporated are assigned numbers starting with the next highest number in the folder. If the specified (or default) folder doesn't exist, the user will be queried prior to its creation. As the messages are processed, a *scan* listing of the new mail is produced.

If the user's profile contains a "Msg—Protect: nnn" entry, it will be used as the protection on the newly created messages, otherwise the MH default of 664 will be used. During all operations on messages, this initially assigned protection will be preserved for each message, so *chmod*(I) may be used to set a protection on an individual message, and its protection will be preserved thereafter.

If the switch '—audit audit-file' is specified (usually as a default switch in the profile), then *inc* will append a header line and a line per message to the end of the specified audit-file with the format:

```
inc date
        <scan line for first message>
        <scan line for second message>
                <etc.>
```

This is useful for keeping track of volume and source of incoming mail. Eventually, *repl*, *forw*, *comp*, and *dist* may also produce audits to this (or another) file, perhaps with "Message-Id:" information to keep an exact correspondence history. "Audit-file" will be in the user's MH directory unless a full path is specified.

Inc will incorporate even illegally formatted messages into the user's MH folder, inserting a blank line prior to the offending component and printing a comment identifying the bad message.

In all cases, the mail file will be zeroed.

FILES

$HOME/mh_profile	The user profile
$HOME/mail	The user's mail drop
<mh-dir>/audit-file	Audit trace file (optional)

PROFILE COMPONENTS

Path:	To determine the user's MH directory
Folder—Protect:	For protection on new folders
Msg—Protect:	For protection on new messages

DEFAULTS

'+folder' defaults to "inbox"

CONTEXT

The folder into which the message is being incorporated will become the current folder, and the first message incorporated will be the current message. This leaves the context ready for a *show* of the first new message.

NAME

jot — print sequential or random data

SYNOPSIS

jot [options] [reps [begin [end [s]]]]

DESCRIPTION

Jot may be used to print out increasing, decreasing, random, or redundant data, usually numbers, one per line. The *options* are understood as follows.

—r Generate random data instead of sequential data, the default.

—b word
Just print *word* repetitively.

—w word
Print *word* with the generated data appended to it. Octal, hexadecimal, exponential, ASCII, zero padded, and right-adjusted representations are possible by using the appropriate *printf*(3) conversion specification inside *word*, in which case the data are inserted rather than appended.

—c This is an abbreviation for **—w %c**.

—s string
Print data separated by *string*. Normally, newlines separate data.

—p precision
Print only as many digits or characters of the data as indicated by the integer *precision*. In the absence of **—p**, the precision is the greater of the precisions of *begin* and *end*. The **—p** option is overridden by whatever appears in a *printf*(3) conversion following **—w**.

The last four arguments indicate, respectively, the number of data, the lower bound, the upper bound, and the step size or, for random data, the seed. While at least one of them must appear, any of the other three may be omitted, and will be considered as such if given as —. Any three of these arguments determines the fourth. If four are specified and the given and computed values of *reps* conflict, the lower value is used. If fewer than three are specified, defaults are assigned left to right, except for *s*, which assumes its default unless both *begin* and *end* are given.

Defaults for the four arguments are, respectively, 100, 1, 100, and 1, except that when random data are requested, *s* defaults to a seed depending upon the time of day. *Reps* is expected to be an unsigned integer, and if given as zero is taken to be infinite. *Begin* and *end* may be given as real numbers or as characters representing the corresponding value in ASCII. The last argument must be a real number.

Random numbers are obtained through *rand*(3). The name *jot* derives in part from *iota*, a function in APL.

EXAMPLES

The command

jot 21 —1 1.00

prints 21 evenly spaced numbers increasing from —1 to 1. The ASCII character set is generated with

jot —c 128 0

and the strings xaa through xaz with

jot —w xa%c 26 a

while 20 random 8-letter strings are produced with

jot —r —c 160 a z | rs —g 0 8

Infinitely many *yes*'s may be obtained through

jot —b yes 0

and thirty *ed*(1) substitution commands applying to lines 2, 7, 12, etc. is the result of

jot —w %ds/old/new/ 30 2 — 5

The stuttering sequence 9, 9, 8, 8, 7, etc. can be produced by suitable choice of precision and step size, as in

jot 0 9 — —.5

and a file containing exactly 1024 bytes is created with

jot —b x 512 > block

Finally, to set tabs four spaces apart starting from column 10 and ending in column 132, use

expand —`jot —s, — 10 132 4`

and to print all lines longer than 90 characters,

grep `jot —s "" —b . 90`.*

SEE ALSO

rs(1), ed(1), yes(1), printf(3), rand(3), expand(1)

AUTHOR

John Kunze

BUGS

NAME

lam — laminate files

SYNOPSIS

lam [—[fp] min.max] [—s sepstring] file ...

DESCRIPTION

Lam copies the named files side by side onto the standard output. Input lines from each *file* become fragments which are assembled into long output lines. The name '—' means the standard input, and may be repeated.

Normally, each option affects only the *file* after it. If the option letter is capitalized it affects all subsequent files until it appears again uncapitalized. The options are described below.

—f min.max
: Print line fragments according to *min.max*, where *min* is the minimum field width and *max* the maximum field width. If *min* begins with a zero, zeros will be added to make up the field width, and if it begins with a '—', the fragment will be left-adjusted within the field.

—p min.max
: Like **—f**, but pad this file's field when end-of-file is reached and other files are still active.

—s sepstring
: Print *sepstring* before printing line fragments from the next file. This option may appear after the last file.

To print files simultaneously for easy viewing use *pr*(1).

EXAMPLES

The command

```
lam file1 file2 file3 file4
```

joins 4 files together along each line. To merge the lines from four different files use

```
lam file1 —S "\
" file2 file3 file4
```

Every 2 lines of a file may be joined on one line with

```
lam — — < file
```

SEE ALSO

pr(1), join(1), printf(3)

AUTHOR

John Kunze

BUGS

NAME

merge — three-way file merge

SYNOPSIS

merge [**-p**] file1 file2 file3

DESCRIPTION

Merge incorporates all changes that lead form *file2* to *file3* into *file1*. The result goes to std. output if **-p** is present, into *file1* otherwise. *Merge* is useful for combining separate changes to an original. Suppose *file2* is the original, and both *file1* and *file3* are modifications of *file2*. Then *merge* combines both changes.

An overlap occurs if both *file1* and *file3* have changes in a common segment of lines. *Merge* prints how many overlaps occurred, and includes both alternatives in the result. The alternatives are delimited as follows:

```
<<<<<<< file1
lines in file1
=======
lines in file3
>>>>>>> file3
```

If there are overlaps, the user should edit the result and delete one of the alternatives.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.0 ; Release Date: 82/11/25 .

SEE ALSO

diff3 (1), diff (1), rcsmerge (1), co (1).

NAME

next — show the next message

SYNOPSIS

next [+folder] [—switches for *l*] [—help]

DESCRIPTION

Next performs a *show* on the next message in the specified (or current) folder. Like *show*, it passes any switches on to the program *l*, which is called to list the message. This command is exactly equivalent to "show next".

FILES

$HOME/mh_profile The user profile

PROFILE COMPONENTS

Path: To determine the user's MH directory
Current-Folder: To find the default current folder

CONTEXT

If a folder is specified, it will become the current folder, and the message that is shown (i.e., the next message in sequence) will become the current message.

NAME

pick — select messages by content

SYNOPSIS

```
pick ( —cc          ) [—src +folder] [msgs] [—help] [—scan] [—noscan]
       —date          [—show] [—noshow] [—nofile] [—nokeep]
       —from
       —search      ) pattern
       —subject
       —to          ) [—file [—preserve] [—link] +folder ... [—nopreserve] [—nolink] ]
       ——component    [—keep [—stay] [—nostay] [+folder ...] ]
```

typically:

```
pick —from jones —scan
pick —to holloway
pick —subject ned —scan —keep
```

DESCRIPTION

Pick searches messages within a folder for the specified contents, then performs several operations on the selected messages.

A modified *grep*(I) is used to perform the searching, so the full regular expression (see *ed*(I)) facility is available within 'pattern'. With '—search', pattern is used directly, and with the others, the grep pattern constructed is:

"^component:pattern"

This means that the pattern specified for a '—search' will be found everywhere in the message, including the header and the body, while the other search requests are limited to the single specified component. The expression '——component pattern' is a shorthand for specifying '—search "component:pattern" '; it is used to pick a component not in the set [cc date from subject to]. An example is "pick ——reply—to pooh —show".

Searching is performed on a per-line basis. Within the header of the message, each component is treated as one long line, but in the body, each line is separate. Lower-case letters in the search pattern will match either lower or upper case in the message, while upper case will match only upper case.

Once the search has been performed, the selected messages are scanned (see *scan*) if the '—scan' switch is given, and then they are shown (see *show*) if the '—show' switch is given. After these two operations, the file operations (if requested) are performed.

The '—file' switch operates exactly like the *file* command, with the same meaning for the '—preserve' and '—link' switches.

The '—keep' switch is similar to '—file', but it produces a folder that is a subfolder of the folder being searched and defines it as the current folder (unless the '—stay' flag is used). This subfolder contains the messages which matched the search criteria. All of the MH commands may be used with the sub-folder as the current folder. This gives the user considerable power in dealing with subsets of messages in a folder.

The messages in a folder produced by '—keep' will always have the same numbers as they have in the source folder (i.e., the '—preserve' switch is automatic). This way, the message numbers are consistent with the folder from which the messages were selected. Messages are not removed from the source folder (i.e., the '—link' switch is assumed). If a '+folder' is not specified, the standard name "select" will be used. (This is the meaning of "(select)" when it appears in the output of the *folder* command.) If '+folder' arguments are given to '—keep', they will be used rather than "select" for the names of the subfolders. This

allows for several subfolders to be maintained concurrently.

When a '—keep' is performed, the subfolder becomes the current folder. This can be overridden by use of the '—stay' switch.

Here's an example:

```
 1 % folder +inbox
 2       inbox+ has  16 messages (  3— 22); cur=  3.
 3 % pick —from dcrocker
 4 6 hits.
 5 [+inbox/select now current]
 6 % folder
 7  inbox/select+ has   6 messages (  3— 16); cur=  3.
 8 % scan
 9   3+ 6/20  Dcrocker       Re: ned file update issue...
10   6  6/23  Dcrocker       removal of files from /tm...
11   8  6/27  Dcrocker       Problems with the new ned...
12  13  6/28  dcrocker       newest nned  I would ap...
13  15  7/ 5  Dcrocker       nned  Last week I asked...
14  16  7/ 5  dcrocker       message id format  I re...
15 % show all | print
16   [produce a full listing of this set of messages on the line printer.]
17 % folder —up
18        inbox+ has  16 messages (  3— 22); cur=  3; (select).
19 % folder —down
20  inbox/select+ has   6 messages (  3— 16); cur=  3.
21 % rmf
22 [+inbox now current]
23 % folder
24        inbox+ has  16 messages (  3— 22); cur=  3.
```

This is a rather lengthy example, but it shows the power of the MH package. In item 1, the current folder is set to inbox. In 3, all of the messages from dcrocker are found in inbox and linked into the folder "inbox/select". (Since no action switch is specified, '—keep' is assumed.) Items 6 and 7 show that this subfolder is now the current folder. Items 8 through 14 are a *scan* of the selected messages (note that they are all from dcrocker and are all in upper and lower case). Item 15 lists all of the messages to the high-speed printer. Item 17 directs *folder* to set the current folder to the parent of the selection-list folder, which is now current. Item 18 shows that this has been done. Item 19 resets the current folder to the selection list, and 21 removes the selection-list folder and resets the current folder to the parent folder, as shown in 22 and 23.

FILES

$HOME/mh_profile	The user profile

PROFILE COMPONENTS

Path:	To determine the user's MH directory
Folder—Protect:	For protection on new folders
Current-Folder:	To find the default current folder

DEFAULTS

'—src +folder' defaults to current
'msgs' defaults to all
'—keep +select' is the default if no '—scan', '—show', or '—file' is specified

CONTEXT

If a '—src +folder' is specified, it will become the current folder, unless a '—keep' with 0 or 1 folder arguments makes the selection-list subfolder the current folder. Each selection-list folder will have its current message set to the first of the messages linked into it unless the selection list already existed, in which case the current message won't be changed.

NAME

prev — show the previous message

SYNOPSIS

prev [+folder] [—switches for *l*] [—help]

DESCRIPTION

Prev performs a *show* on the previous message in the specified (or current) folder. Like *show*, it passes any switches on to the program *l*, which is called to list the message. This command is exactly equivalent to "show prev".

FILES

$HOME/mh_profile The user profile

PROFILE COMPONENTS

Path: To determine the user's MH directory
Current-Folder: To find the default current folder

CONTEXT

If a folder is specified, it will become current, and the message that is shown (i.e., the previous message in sequence) will become the current message.

NAME

prmdir — remove a project directory

SYNOPSIS

prmdir [—**fru**] [{+—}**T** type[,type ...]] pdirname ...

DESCRIPTION

Prmdir deletes a project directory called *pdirname*. The directory must be empty.

If the —**r** option is specified, *prmdir* recursively deletes the entire contents of a project directory, and the directory itself. The user is asked to confirm the generated *rm —r* command before the directory is deleted. Subdirectories that are project root directories must be removed using *rmproject* before attempting to remove *pdirname*. Write permission is required in all subdirectories.

Prmdir may also be used to convert an existing project directory to a regular directory using the —**u** option.

OPTIONS

—**f** Stands for force. No questions are asked. This option overrides any mode restrictions.

—**r** Recursively remove project directories.

—**u** Undefine a project directory and convert it to a regular directory.

—**T** *type*
Remove a type label from a project directory.

FILES

...	Project link directory.
..._temp	Temporary project link directory.

SEE ALSO

pmkdir(1P), rm(1), rmdir(1), rmproject(1P)

DIAGNOSTICS

The error message, "prmdir: *project/...* temporarily unavailable", indicates that a '..._temp' temporary project link directory exists. This could be because another user is altering the project link directory, or because a system crash terminated *prmdir* prematurely. If the latter case, then removing the temporary file will fix the problem.

Exit status 0 is normal. Exit status 1 indicates an error.

AUTHOR

Peter J. Nicklin

BUGS

If a project directory has already been removed by the *rmdir* or *rm -r* commands, that directory must be recreated using *mkdir* before *prmdir* will remove the directory from the project.

NAME

prompter — prompting editor front end

SYNOPSIS

prompter [—erase chr] [—kill chr] [—help]

DESCRIPTION

This program is not called directly but takes the place of an editor and acts as an editor front end. *Prompter* is an editor which allows rapid composition of messages. It is particularly useful to network and low-speed (less than 2400 baud) users of MH. It is an MH program in that it can have its own profile entry with switches, but it can't be invoked directly as all other MH commands can; it is an editor in that it is invoked by an "—editor prompter" switch or by the profile entry "Editor: prompter", but functionally it is merely a text-collector and not a true editor.

Prompter expects to be called from *comp*, *repl*, *dist*, or *forw*, with a draft file as an argument. For example, "comp —editor prompter" will call *prompter* with the file "draft" already set up with blank components. For each blank component it finds in the draft, it prompts the user and accepts a response. A <RETURN> will cause the whole component to be left out. A "\" preceding a <RETURN> will continue the response on the next line, allowing for multiline components.

Any component that is non-blank will be copied and echoed to the terminal.

The start of the message body is prompted by a line of dashes. If the body is non-blank, the prompt is "———Enter additional text". Message-body typing is terminated with a <CTRL-D> (or <OPEN>). Control is returned to the calling program, where the user is asked "What now?". See *comp* for the valid options.

The line editing characters for kill and erase may be specified by the user via the arguments "—kill chr" and "—erase chr", where chr may be a character; or "\nnn", where nnn is the octal value for the character. (Again, these may come from the default switches specified in the user's profile.)

A <DEL> during message-body typing is equivalent to <CTRL-D> for compatibility with NED. A <DEL> during component typing will abort the command that invoked *prompter*.

PROFILE COMPONENTS

prompter-next: editor to be used on exit from *prompter*

NAME

rcs — change RCS file attributes

SYNOPSIS

rcs [options] file ...

DESCRIPTION

Rcs creates new RCS files or changes attributes of existing ones. An RCS file contains multiple revisions of text, an access list, a change log, descriptive text, and some control attributes. For *rcs* to work, the caller's login name must be on the access list, except if the access list is empty, the caller is the owner of the file or the superuser, or the -i option is present.

Files ending in ',v' are RCS files, all others are working files. If a working file is given, *rcs* tries to find the corresponding RCS file first in directory ./RCS and then in the current directory, as explained in *co* (1).

—i creates and initializes a new RCS file, but does not deposit any revision. If the RCS file has no path prefix, *rcs* tries to place it first into the subdirectory ./RCS, and then into the current directory. If the RCS file already exists, an error message is printed.

—a*logins* appends the login names appearing in the comma-separated list *logins* to the access list of the RCS file.

—A*oldfile* appends the access list of *oldfile* to the access list of the RCS file.

—e[*logins*] erases the login names appearing in the comma-separated list *logins* from the access list of the RCS file. If *logins* is omitted, the entire access list is erased.

—c*string* sets the comment leader to *string*. The comment leader is printed before every log message line generated by the keyword Log during checkout (see *co*). This is useful for programming languages without multi-line comments. During *rcs -i* or initial *ci*, the comment leader is guessed from the suffix of the working file.

—l[*rev*] locks the revision with number *rev*. If a branch is given, the latest revision on that branch is locked. If *rev* is omitted, the latest revision on the trunk is locked. Locking prevents overlapping changes. A lock is removed with *ci* or *rcs -u* (see below).

—u[*rev*] unlocks the revision with number *rev*. If a branch is given, the latest revision on that branch is unlocked. If *rev* is omitted, the latest lock held by the caller is removed. Normally, only the locker of a revision may unlock it. Somebody else unlocking a revision breaks the lock. This causes a mail message to be sent to the original locker. The message contains a commentary solicited from the breaker. The commentary is terminated with a line containing a single '.' or control-D.

—L sets locking to *strict*. Strict locking means that the owner of an RCS file is not exempt from locking for checkin. This option should be used for files that are shared.

—U sets locking to non-strict. Non-strict locking means that the owner of a file need not lock a revision for checkin. This option should NOT be used for files that are shared. The default (-L or -U) is determined by your system administrator.

—n*name*[:*rev*]

associates the symbolic name *name* with the branch or revision *rev*. *Rcs* prints an error message if *name* is already associated with another number. If *rev* is omitted, the symbolic name is deleted.

—N*name*[:*rev*]

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

—o*range* deletes ("outdates") the revisions given by *range*. A range consisting of a single revision number means that revision. A range consisting of a branch number means the latest revision on that branch. A range of the form *rev1—rev2* means revisions *rev1* to *rev2* on the same branch, *—rev* means from the beginning of the branch containing *rev* up to and including *rev*, and *rev—* means from revision *rev* to the end of the branch containing *rev*. None of the outdated revisions may have branches or locks.

—q quiet mode; diagnostics are not printed.

—s*state*[:*rev*]
sets the state attribute of the revision *rev* to *state*. If *rev* is omitted, the latest revision on the trunk is assumed; If *rev* is a branch number, the latest revision on that branch is assumed. Any identifier is acceptable for *state*. A useful set of states is *Exp* (for experimental), *Stab* (for stable), and *Rel* (for released). By default, *ci* sets the state of a revision to *Exp*.

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

DIAGNOSTICS

The RCS file name and the revisions outdated are written to the diagnostic output. The exit status always refers to the last RCS file operated upon, and is 0 if the operation was successful, 1 otherwise.

FILES

The caller of the command must have read/write permission for the directory containing the RCS file and read permission for the RCS file itself. *Rcs* creates a semaphore file in the same directory as the RCS file to prevent simultaneous update. For changes, *rcs* always creates a new file. On successful completion, *rcs* deletes the old one and renames the new one. This strategy makes links to RCS files useless.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.1 ; Release Date: 83/04/04 .

SEE ALSO

co (1), ci (1), ident(1), rcsdiff (1), rcsintro (1), rcsmerge (1), rlog (1), rcsfile (5), sccstorcs (8).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

NAME

rcsdiff — compare RCS revisions

SYNOPSIS

rcsdiff [**-b**] [**-cefhn**] [**-r***rev1*] [**-r***rev2*] file ...

DESCRIPTION

Rcsdiff runs *diff* (1) to compare two revisions of each RCS file given. A file name ending in ',v' is an RCS file name, otherwise a working file name. *Rcsdiff* derives the working file name from the RCS file name and vice versa, as explained in *co* (1). Pairs consisting of both an RCS and a working file name may also be specified.

The options **-b**, **-c**, **-e**, **-f**, and **-h** have the same effect as described in *diff* (1); option **-n** generates an edit script of the format used by RCS.

If both *rev1* and *rev2* are omitted, *rcsdiff* compares the latest revision on the trunk with the contents of the corresponding working file. This is useful for determining what you changed since the last checkin.

If *rev1* is given, but *rev2* is omitted, *rcsdiff* compares revision *rev1* of the RCS file with the contents of the corresponding working file.

If both *rev1* and *rev2* are given, *rcsdiff* compares revisions *rev1* and *rev2* of the RCS file.

Both *rev1* and *rev2* may be given numerically or symbolically.

EXAMPLES

The command

```
rcsdiff f.c
```

runs *diff* on the latest trunk revision of RCS file f.c,v and the contents of working file f.c.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.0 ; Release Date: 83/01/15 .

SEE ALSO

ci (1), co (1), diff (1), ident (1), rcs (1), rcsintro (1), rcsmerge (1), rlog (1), rcsfile (5).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

NAME

rcsintro - introduction to RCS commands

DESCRIPTION

The Revision Control System (RCS) manages multiple revisions of text files. RCS automates the storing, retrieval, logging, identification, and merging of revisions. RCS is useful for text that is revised frequently, for example programs, documentation, graphics, papers, form letters, etc.

The basic user interface is extremely simple. The novice only needs to learn two commands: *ci* and *co*. *Ci*, short for "checkin", deposits the contents of a text file into an archival file called an RCS file. An RCS file contains all revisions of a particular text file. *Co*, short for "checkout", retrieves revisions from an RCS file.

Functions of RCS

- Storage and retrieval of multiple revisions of text. RCS saves all old revisions in a space efficient way. Changes no longer destroy the original, because the previous revisions remain accessible. Revisions can be retrieved according to ranges of revision numbers, symbolic names, dates, authors, and states.
- Maintenance of a complete history of changes. RCS logs all changes automatically. Besides the text of each revision, RCS stores the author, the date and time of checkin, and a log message summarizing the change. The logging makes it easy to find out what happened to a module, without having to compare source listings or having to track down colleagues.
- Resolution of access conflicts. When two or more programmers wish to modify the same revision, RCS alerts the programmers and prevents one modification from corrupting the other.
- Maintenance of a tree of Revisions. RCS can maintain separate lines of development for each module. It stores a tree structure that represents the ancestral relationships among revisions.
- Merging of revisions and resolution of conflicts. Two separate lines of development of a module can be coalesced by merging. If the revisions to be merged affect the same sections of code, RCS alerts the user about the overlapping changes.
- Release and configuration control. Revisions can be assigned symbolic names and marked as released, stable, experimental, etc. With these facilities, configurations of modules can be described simply and directly.
- Automatic identification of each revision with name, revision number, creation time, author, etc. The identification is like a stamp that can be embedded at an appropriate place in the text of a revision. The identification makes it simple to determine which revisions of which modules make up a given configuration.
- Minimization of secondary storage. RCS needs little extra space for the revisions (only the differences). If intermediate revisions are deleted, the corresponding deltas are compressed accordingly.

Getting Started with RCS

Suppose you have a file f.c that you wish to put under control of RCS. Invoke the checkin command

```
ci f.c
```

This command creates the RCS file f.c,v, stores f.c into it as revision 1.1, and deletes f.c. It also asks you for a description. The description should be a synopsis of the contents of the

file. All later checkin commands will ask you for a log entry, which should summarize the changes that you made.

Files ending in ,v are called RCS files ('v' stands for 'versions'), the others are called working files. To get back the working file f.c in the previous example, use the checkout command

co f.c

This command extracts the latest revision from f.c,v and writes it into f.c. You can now edit f.c and check it back in by invoking

ci f.c

Ci increments the revision number properly. If *ci* complains with the message

ci error: no lock set by <your login>

then your system administrator has decided to create all RCS files with the locking attribute set to 'strict'. In this case, you should have locked the revision during the previous checkout. Your last checkout should have been

co -l f.c

Of course, it is too late now to do the checkout with locking, because you probably modified f.c already, and a second checkout would overwrite your modifications. Instead, invoke

rcs -l f.c

This command will lock the latest revision for you, unless somebody else got ahead of you already. In this case, you'll have to negotiate with that person.

Locking assures that you, and only you, can check in the next update, and avoids nasty problems if several people work on the same file. Even if a revision is locked, it can still be checked out for reading, compiling, etc. All that locking prevents is a CHECKIN by anybody but the locker.

If your RCS file is private, i.e., if you are the only person who is going to deposit revisions into it, strict locking is not needed and you can turn it off. If strict locking is turned off, the owner of the RCS file need not have a lock for checkin; all others still do. Turning strict locking off and on is done with the commands

rcs -U f.c and rcs -L f.c

If you don't want to clutter your working directory with RCS files, create a subdirectory called RCS in your working directory, and move all your RCS files there. RCS commands will look first into that directory to find needed files. All the commands discussed above will still work, without any modification. (Actually, pairs of RCS and working files can be specified in 3 ways: (a) both are given, (b) only the working file is given, (c) only the RCS file is given. Both RCS and working files may have arbitrary path prefixes; RCS commands pair them up intelligently).

To avoid the deletion of the working file during checkin (in case you want to continue editing), invoke

ci -l f.c or ci -u f.c

These commands check in f.c as usual, but perform an implicit checkout. The first form also locks the checked in revision, the second one doesn't. Thus, these options save you one checkout operation. The first form is useful if locking is strict, the second one if not strict. Both update the identification markers in your working file (see below).

You can give *ci* the number you want assigned to a checked in revision. Assume all your revisions were numbered 1.1, 1.2, 1.3, etc., and you would like to start release 2. The command

```
ci -r2 f.c      or      ci -r2.1 f.c
```

assigns the number 2.1 to the new revision. From then on, *ci* will number the subsequent revisions with 2.2, 2.3, etc. The corresponding *co* commands

```
co -r2 f.c      and     co -r2.1 f.c
```

retrieve the latest revision numbered 2.x and the revision 2.1, respectively. *Co* without a revision number selects the latest revision on the "trunk", i.e., the highest revision with a number consisting of 2 fields. Numbers with more than 2 fields are needed for branches. For example, to start a branch at revision 1.3, invoke

```
ci -r1.3.1 f.c
```

This command starts a branch numbered 1 at revision 1.3, and assigns the number 1.3.1.1 to the new revision. For more information about branches, see *rcsfile*(5).

Automatic Identification

RCS can put special strings for identification into your source and object code. To obtain such identification, place the marker

```
$Header$
```

into your text, for instance inside a comment. RCS will replace this marker with a string of the form

```
$Header: filename revision_number date time author state $
```

With such a marker on the first page of each module, you can always see with which revision you are working. RCS keeps the markers up to date automatically. To propagate the markers into your object code, simply put them into literal character strings. In C, this is done as follows:

```
static char rcsid[] = "$Header$";
```

The command *ident* extracts such markers from any file, even object code and dumps. Thus, *ident* lets you find out which revisions of which modules were used in a given program.

You may also find it useful to put the marker Log into your text, inside a comment. This marker accumulates the log messages that are requested during checkin. Thus, you can maintain the complete history of your file directly inside it. There are several additional identification markers; see *co*(1) for details.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.0 ; Release Date: 83/05/11 .

SEE ALSO

ci(1), co(1), ident(1), merge(1), rcs(1), rcsdiff(1), rcsmerge(1), rlog(1), rcsfile(5).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

rcsmerge — merge RCS revisions

SYNOPSIS

rcsmerge -r*rev1* [**-r***rev2*] [**-p**] file

DESCRIPTION

Rcsmerge incorporates the changes between *rev1* and *rev2* of an RCS file into the corresponding working file. If **-p** is given, the result is printed on the std. output, otherwise the result overwrites the working file.

A file name ending in ',v' is an RCS file name, otherwise a working file name. *Merge* derives the working file name from the RCS file name and vice versa, as explained in *co* (1). A pair consisting of both an RCS and a working file name may also be specified.

Rev1 may not be omitted. If *rev2* is omitted, the latest revision on the trunk is assumed. Both *rev1* and *rev2* may be given numerically or symbolically.

Rcsmerge prints a warning if there are overlaps, and delimits the overlapping regions as explained in *co -j*. The command is useful for incorporating changes into a checked-out revision.

EXAMPLES

Suppose you have released revision 2.8 of f.c. Assume furthermore that you just completed revision 3.4, when you receive updates to release 2.8 from someone else. To combine the updates to 2.8 and your changes between 2.8 and 3.4, put the updates to 2.8 into file f.c and execute

```
rcsmerge -p -r2.8 -r3.4 f.c >f.merged.c
```

Then examine f.merged.c. Alternatively, if you want to save the updates to 2.8 in the RCS file, check them in as revision 2.8.1.1 and execute *co -j*:

```
ci -r2.8.1.1 f.c
co -r3.4 -j2.8:2.8.1.1 f.c
```

As another example, the following command undoes the changes between revision 2.4 and 2.8 in your currently checked out revision in f.c.

```
rcsmerge -r2.8 -r2.4 f.c
```

Note the order of the arguments, and that f.c will be overwritten.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.0 ; Release Date: 83/01/15 .

SEE ALSO

ci (1), co (1), merge (1), ident (1), rcs (1), rcsdiff (1), rlog (1), rcsfile (5).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

Rcsmerge does not work for files that contain lines with a single '.'.

NAME

repl — reply to a message

SYNOPSIS

repl [+folder] [msg] [—editor editor] [—inplace] [—annotate] [—help] [—noinplace] [—noannotate]

DESCRIPTION

Repl aids a user in producing a reply to an existing message. In its simplest form (with no arguments), it will set up a message-form skeleton in reply to the current message in the current folder, invoke the editor, and send the composed message if so directed. The composed message is constructed as follows:

To: <Reply-To> or <From>
cc: <cc>, <To>
Subject: Re: <Subject>
In-reply-to: Your message of <Date>
<Message-Id>

where field names enclosed in angle brackets (< >) indicate the contents of the named field from the message to which the reply is being made. Once the skeleton is constructed, an editor is invoked (as in *comp*, *dist*, and *forw*). While in the editor, the message being replied to is available through a link named "@". In NED, this means the replied-to message may be "used" with "use @", or put in a window by "window @".

As in *comp*, *dist*, and *forw*, the user will be queried before the message is sent. If '—annotate' is specified, the replied-to message will be annotated with the single line

Replied: Date.

The command "comp —use" may be used to pick up interrupted editing, as in *dist* and *forw*; the '—inplace' switch annotates the message in place, so that all folders with links to it will see the annotation.

FILES

$HOME/mh_profile	The user profile
<mh-dir>/draft	The constructed message file
/usr/bin/send	To send the composed message

PROFILE COMPONENTS

Path:	To determine the user's MH directory
Editor:	To override the use of /bin/ned as the default editor
Current-Folder:	To find the default current folder

DEFAULTS

'+folder' defaults to current
'msgs' defaults to cur
'—editor' defaults to /bin/ned
'—noannotate'
'—noinplace'

CONTEXT

If a '+folder' is specified, it will become the current folder, and the current message will be set to the replied-to message.

NAME

rlog — print log messages and other information about RCS files

SYNOPSIS

rlog [options] file ...

DESCRIPTION

Rlog prints information about RCS files. Files ending in ',v' are RCS files, all others are working files. If a working file is given, *rlog* tries to find the corresponding RCS file first in directory ./RCS and then in the current directory, as explained in *co* (1).

Rlog prints the following information for each RCS file: RCS file name, working file name, head (i.e., the number of the latest revision on the trunk), access list, locks, symbolic names, suffix, total number of revisions, number of revisions selected for printing, and descriptive text. This is followed by entries for the selected revisions in reverse chronological order for each branch. For each revision, *rlog* prints revision number, author, date/time, state, number of lines added/deleted (with respect to the previous revision), locker of the revision (if any), and log message. Without options, *rlog* prints complete information. The options below restrict this output.

—L ignores RCS files that have no locks set; convenient in combination with **-R**, **-h**, or **-l**.

—R only prints the name of the RCS file; convenient for translating a working file name into an RCS file name.

—h prints only RCS file name, working file name, head, access list, locks, symbolic names, and suffix.

—t prints the same as **-h**, plus the descriptive text.

—d*dates* prints information about revisions with a checkin date/time in the ranges given by the semicolon-separated list of *dates*. A range of the form *d1*<*d2* or *d2*>*d1* selects the revisions that were deposited between *d1* and *d2*, (inclusive). A range of the form <*d* or *d*> selects all revisions dated *d* or earlier. A range of the form *d*< or >*d* selects all revisions dated *d* or later. A range of the form *d* selects the single, latest revision dated *d* or earlier. The date/time strings *d*, *d1*, and *d2* are in the free format explained in *co* (1). Quoting is normally necessary, especially for < and >. Note that the separator is a semicolon.

—l[*lockers*]
prints information about locked revisions. If the comma-separated list *lockers* of login names is given, only the revisions locked by the given login names are printed. If the list is omitted, all locked revisions are printed.

—r*revisions*
prints information about revisions given in the comma-separated list *revisions* of revisions and ranges. A range *rev1*—*rev2* means revisions *rev1* to *rev2* on the same branch, —*rev* means revisions from the beginning of the branch up to and including *rev*, and *rev*— means revisions starting with *rev* to the end of the branch containing *rev*. An argument that is a branch means all revisions on that branch. A range of branches means all revisions on the branches in that range.

—s*states* prints information about revisions whose state attributes match one of the states given in the comma-separated list *states*.

—w[*logins*]
prints information about revisions checked in by users with login names appearing in the comma-separated list *logins*. If *logins* is omitted, the user's login is assumed.

Rlog prints the intersection of the revisions selected with the options **-d**, **-l**, **-s**, **-w**, intersected with the union of the revisions selected by **-b** and **-r**.

EXAMPLES

```
rlog -L -R RCS/*,v
rlog -L -h RCS/*,v
rlog -L -l RCS/*,v
rlog RCS/*,v
```

The first command prints the names of all RCS files in the subdirectory 'RCS' which have locks. The second command prints the headers of those files, and the third prints the headers plus the log messages of the locked revisions. The last command prints complete information.

DIAGNOSTICS

The exit status always refers to the last RCS file operated upon, and is 0 if the operation was successful, 1 otherwise.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.2 ; Release Date: 83/05/11 .

SEE ALSO

ci (1), co (1), ident(1), rcs (1), rcsdiff (1), rcsintro (1), rcsmerge (1), rcsfile (5), sccstorcs (8).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

NAME

rmf — remove folder

SYNOPSIS

rmf [+folder] [—help]

DESCRIPTION

Rmf removes all of the files (messages) within the specified (or default) folder, and then removes the directory (folder). If there are any files within the folder which are not a part of MH, they will *not* be removed, and an error will be produced. If the folder is given explicitly or the current folder is a subfolder (i.e., a selection list from *pick*), it will be removed without confirmation. If no argument is specified and the current folder is not a selection-list folder, the user will be asked for confirmation.

Rmf irreversibly deletes messages that don't have other links, so use it with caution.

If the folder being removed is a subfolder, the parent folder will become the new current folder, and *rmf* will produce a message telling the user this has happened. This provides an easy mechanism for selecting a set of messages, operating on the list, then removing the list and returning to the current folder from which the list was extracted. (See the example under *pick*.)

The files that *rmf* will delete are cur, any file beginning with a comma, and files with purely numeric names. All others will produce error messages.

Rmf of a read-only folder will delete the "cur—" entry from the profile without affecting the folder itself.

FILES

PROFILE COMPONENTS

$HOME/mh_profile	The user profile
Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

DEFAULTS

'+folder' defaults to current, usually with confirmation

CONTEXT

Rmf will set the current folder to the parent folder if a subfolder is removed; or if the current folder is removed, it will make "inbox" current. Otherwise, it doesn't change the current folder or message.

NAME

rmm — remove messages

SYNOPSIS

rmm [+folder] [msgs] [—help]

DESCRIPTION

Rmm removes the specified messages by renaming the message files with preceding commas. (This is the Rand-UNIX backup file convention.)

The current message is not changed by *rmm*, so a *next* will advance to the next message in the folder as expected.

FILES

$HOME/mh_profile The user profile

PROFILE COMPONENTS

Path: To determine the user's MH directory
Current-Folder: To find the default current folder

DEFAULTS

'+folder' defaults to current
'msgs' defaults to cur

CONTEXT

If a folder is given, it will become current.

NAME

rs — reshape a data array

SYNOPSIS

rs [—[csCS][x][kKgGw][N]tTeEnyjhHm] [rows [cols]]

DESCRIPTION

Rs reads the standard input, interpreting each line as a row of blank-separated entries in an array, transforms the array according to the *options*, and writes it on the standard output. With no arguments it transforms stream input into a columnar format convenient for terminal viewing.

Given positive integers for *rows* and *cols*, the program produces output with the corresponding shape, truncating surplus data and supplying missing data as necessary. If only one of them is a positive integer, then *rs* computes a value for the other which will accomodate all of the data.

The options are described below.

—cx Input columns are delimited by the single character *x*. A missing *x* is taken to be '^I'.

—sx Like **—c**, but maximal strings of *x* are delimiters.

—Cx Output columns are delimited by the single character *x*. A missing *x* is taken to be '^I'.

—Sx Like **—C**, but maximal strings of *x* are delimiters.

—t Fill in the rows of the output array using the columns of the input array, that is, transpose the input while honoring any *rows* and *cols* specifications.

—T Print the pure transpose of the input, ignoring any *rows* or *cols* specification.

—kN Ignore the first *N* lines of input.

—KN Like **—k**, but print the ignored lines.

—gN The gutter width (inter-column space), normally 2, is taken to be *N*.

—GN The gutter width has *N* percent of the maximum column width added to it.

—e Consider each line of input as an array entry.

—n On lines having fewer entries than the first line, use null entries to pad out the line. Normally, missing entries are taken from the next line of input.

—y If there are too few entries to make up the output dimensions, pad the output by recycling the input from the beginning. Normally, the output is padded with blanks.

—h Print the shape of the input array and do nothing else. The shape is just the number of lines and the number of entries on the first line.

—H Like **—h**, but also print the length of each line.

—j Right adjust entries within columns.

—wN The width of the display, normally 80, is taken to be the positive integer *N*.

—m Do not trim excess delimiters from the ends of the output array.

With no arguments, *rs* behaves as if given **—et**. Option letters which take numerical arguments interpret a missing number as zero unless otherwise indicated.

EXAMPLES

Rs can be used as a filter to convert the stream output of several programs (e.g., *spell*, *du*, *file*, *look*, *nm*, *who*, and *wc*(1)) into a convenient "window" format, as in

who | rs

This function has been incorporated into the *ls*(1) program, though for most programs with similar output *rs* suffices.

To convert stream input into vector output and back again, use

rs 1 0 | rs 0 1

A 10 by 10 array of random numbers from 1 to 100 and its transpose can be generated with

jot —r 100 | rs 10 10 | tee array | rs —T > tarray

In the editor *ex*(1), a file consisting of a 3-column table that has undergone insertions and deletions can be neatly reshaped into 3 columns with

:1,$!rs 0 3

Finally, to sort a database by the first line of each 4-line field, try

rs —lC 0 4 | sort | rs —c 0 1

SEE ALSO

jot(1), ex(1), sort(1), pr(1),

AUTHOR

John Kunze

BUGS

Handles only two dimensional arrays.

Fields cannot be defined yet on character positions.

Re-ordering of columns is not yet possible.

There are too many options.

NAME

scan — produce a one-line-per-message scan listing

SYNOPSIS

scan [+folder] [msgs] [—ff] [—header] [—help] [—noff] [—noheader]

DESCRIPTION

Scan produces a one-line-per-message listing of the specified messages. Each *scan* line contains the message number (name), the date, the "From" field, the "Subject" field, and, if room allows, some of the body of the message. For example:

```
^ #~~^Date~~^ From~~^Subject [Body]
^15+~~^7/ 5~~^Dcrocker~~^nned Last week I asked some of
^16 —~~^7/ 5~~^dcrocker~~^message id format I recommend
^18~~^7/ 6~~^Obrien~~^Re: Exit status from mkdir
^19~~^7/ 7~~^Obrien~~^"scan" listing format in MH
```

The '+' on message 15 indicates that it is the current message. The '—' on message 16 indicates that it has been replied to, as indicated by a "Replied:" component produced by an '—annotate' switch to the *repl* command.

If there is sufficient room left on the *scan* line after the subject, the line will be filled with text from the body, preceded by . *Scan* actually reads each of the specified messages and parses them to extract the desired fields. During parsing, appropriate error messages will be produced if there are format errors in any of the messages.

The '—header' switch produces a header line prior to the *scan* listing, and the '—ff' switch will cause a form feed to be output at the end of the *scan* listing.

FILES

^$HOME/mh_profile~~^The user profile

PROFILE COMPONENTS

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

DEFAULTS

'+folder' defaults to current
'msgs' defaults to all
'—noff'
'—noheader'

CONTEXT

If a folder is given, it will become current. The current message is unaffected.

NAME

sccstorcs — build RCS file from SCCS file

SYNOPSIS

sccstorcs [—t] [—v] *s.file* ...

DESCRIPTION

Sccstorcs builds an RCS file from each SCCS file argument. The deltas and comments for each delta are preserved and installed into the new RCS file in order. Also preserved are the user access list and descriptive text, if any, from the SCCS file.

The following flags are meaningful:

—t Trace only. Prints detailed information about the SCCS file and lists the commands that would be executed to produce the RCS file. No commands are actually executed and no RCS file is made.

—v Verbose. Prints each command that is run while it is building the RCS file.

FILES

For each s.*somefile*, *Sccstorcs* writes the files *somefile* and *somefile*,v which should not already exist. *Sccstorcs* will abort, rather than overwrite those files if they do exist.

SEE ALSO

ci (1), co (1), rcs (1).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

DIAGNOSTICS

All diagnostics are written to stderr. Non-zero exit status on error.

BUGS

Sccstorcs does not preserve all SCCS options specified in the SCCS file. Most notably, it does not preserve removed deltas, MR numbers, and cutoff points.

AUTHOR

Ken Greer

NAME

send — send a message

SYNOPSIS

send [file] [—draft] [—verbose] [—format] [—msgid] [—help] [—noverbose] [—noformat] [—nomsgid]

DESCRIPTION

Send will cause the specified file (default <mh-dir>/draft) to be delivered to each of the addresses in the "To:", "cc:", and "Bcc:" fields of the message. If '—verbose' is specified, *send*; will monitor the delivery of local and net mail. *Send* with no argument will query whether the draft is the intended file, whereas '—draft' will suppress this question. Once the message has been mailed (or queued) successfully, the file will be renamed with a leading comma, which allows it to be retreived until the next draft message is sent. If there are errors in the formatting of the message, *send*; will abort with a (hopefully) helpful error message.

If a "Bcc:" field is encountered, its addresses will be used for delivery, but the "Bcc:" field itself will be deleted from all copies of the outgoing message.

Prior to sending the message, the fields "From: user", and "Date: now" will be prepended to the message. If '—msgid' is specified, then a "Message-Id:" field will also be added to the message. If the message already contains a "From:" field, then a "Sender: user" field will be added instead. (An already existing "Sender:" field will be deleted from the message.)

If the user doesn't specify '—noformat', each of the entries in the "To:" and "cc:" fields will be replaced with "standard" format entries. This standard format is designed to be usable by all of the message handlers on the various systems around the ARPANET.

If an "Fcc: folder" is encountered, the message will be copied to the specified folder in the format in which it will appear to any receivers of the message. That is, it will have the prepended fields and field reformatting.

If a "Distribute-To:" field is encountered, the message is handled as a redistribution message (see *dist* for details), with "Distribution-Date: now" and "Distribution-From: user" added.

FILES

$HOME/mh_profile The user profile

PROFILE COMPONENTS

Path: To determine the user's MH directory

DEFAULTS

'file' defaults to draft
'—noverbose'
'—format'
'—nomsgid'

CONTEXT

Send has no effect on the current message or folder.

NAME

show — show (list) messages

SYNOPSIS

show [+folder] [msgs] [—pr] [—nopr] [—draft] [—help] [*l* or *pr* switches]

DESCRIPTION

Show lists each of the specified messages to the standard output (typically, the terminal). The messages are listed exactly as they are, with no reformatting. A program called *l* is invoked to do the listing, and any switches not recognized by *show* are passed along to *l*.

If no "msgs" are specified, the current message is used. If more than one message is specified, *l* will prompt for a <return> prior to listing each message.

l will list each message, a page at a time. When the end of page is reached, *l* will ring the bell and wait for a <RETURN> or <CTRL-D>. If a <return> is entered, *l* will clear the screen before listing the next page, whereas <CTRL-D> will not. The switches to *l* are '—p#' to indicate the page length in lines, and '—w#' to indicate the width of the page in characters.

If the standard output is not a terminal, no queries are made, and each file is listed with a one-line header and two lines of separation.

If '—pr' is specified, then *pr*(I) will be invoked rather than *l*, and the switches (other than '—draft') will be passed along. "Show —draft" will list the file <mh-dir>/draft if it exists.

FILES

$HOME/mh_profile	The user profile
/bin/l	Screen-at-a-time list program
/bin/pr	*pr*(1)

PROFILE COMPONENTS

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

DEFAULTS

'+folder' defaults to current
'msgs' defaults to cur
'—nopr'

CONTEXT

If a folder is given, it will become the current message. The last message listed will become the current message.

MH

A Mail Handling System

for UNIX

October, 1979

Bruce Borden

The Rand Corporation
1700 Main Street
Santa Monica, CA 90406

(213) 399-0568 x 7463

PREFACE

This report describes a system for dealing with messages transmitted on a computer. Such messages might originate with other users of the same computer or might come from an outside source through a network to which the user's computer is connected. Such computer-based message systems are becoming increasingly widely used, both within and outside the Department of Defense.

The message handling system MH was developed for two reasons. One was to investigate some research ideas concerning how a message system might take advantage of the architecture of the UNIX time-sharing operating system for Digital Equipment Corporation PDP-11 and VAX computers, and the special features of UNIX's command-level interface with the user (the "shell"). The other reason was to provide a better and more adaptable base than that of conventional designs on which to build a command and control message system. The effort has succeeded in both regards, although this report mainly describes the message system itself and how it fits in with UNIX. The main research results are being described and analyzed in a forthcoming Rand report. The system is currently being used as part of a tactical command and control "laboratory," which is also being described in a separate report.

The present report should be of interest to three groups of readers. First, it is a complete reference manual for the users of MH (although users outside of Rand must take into account differences from the local Rand operating system). Second, it should be of interest to those who have a general knowledge of computer-based message systems, both in civilian and military applications. Finally, it should be of interest to those who build large subsystems that interface with users, since it illustrates a new approach to such interfaces.

The MH system was developed by the first author, using an approach suggested by the other two authors. Valuable assistance was provided by Phyllis Kantar in the later stages of the system's implementation. Several colleagues contributed to the ideas included in this system, particularly Robert Anderson and David Crocker. In addition, valuable experience in message systems, and a valuable source of ideas, was available to us in the form of a previous message system for UNIX called MS, designed at Rand by David Crocker.

This report was prepared as part of the Rand project entitled "Data Automation Research", sponsored by Project AIR FORCE.

CONTENTS

SUMMARY

Electronic communication of text messages is becoming commonplace. Computer-based message systems—software packages that provide tools for dealing with messages—are used in many contexts. In particular, message systems are becoming increasingly important in command and control and intelligence applications.

This report describes a message handling system called MH. This system provides the user with tools to compose, send, receive, store, retrieve, forward, and reply to messages. MH has been built on the UNIX time-sharing system, a popular operating system developed for the DEC PDP-11 and VAX classes of computers.

A complete description of MH is given for users of the system. For those who do not intend to use the system, this description gives a general idea of what a message system is like. The system involves some new ideas about how large subsystems can be constructed. These design concepts and a comparison of MH with other message systems will be published in a forthcoming Rand report.

The interesting and unusual features of MH include the following: The user command interface to MH is the UNIX "shell" (the standard UNIX command interpreter). Each separable component of message handling, such as message composition or message display, is a separate command. Each program is driven from and updates a private user environment, which is stored as a file between program invocations. This private environment also contains information to "custom tailor" MH to the individual's tastes. MH stores each message as a separate file under UNIX, and it utilizes the tree-structured UNIX file system to organize groups of files within separate directories or "folders." All of the UNIX facilities for dealing with files and directories, such as renaming, copying, deleting, cataloging, off-line printing, etc., are applicable to messages and directories of messages (folders). Thus, important capabilities needed in a message system are available in MH without the need (often seen in other message systems) for code that duplicates the facilities of the supporting operating system. It also allows users familiar with the shell to use MH with minimal effort.

1. INTRODUCTION

Although people can travel cross-country in hours and can reach others by telephone in seconds, communications still depend heavily upon paper, most of which is distributed through the mails.

There are several major reasons for this continued dependence on written documents. First, a written document may be proofread and corrected prior to its distribution, giving the author complete control over his words. Thus, a written document is better than a telephone conversation in this respect. Second, a carefully written document is far less likely to be misinterpreted or poorly translated than a phone conversation. Third, a signature offers reasonable verification of authorship, which cannot be provided with media such as telegrams.

However, the need for fast, accurate, and reproducible document distribution is obvious. One solution in widespread use is the telefax. Another that is rapidly gaining popularity is electronic mail. Electronic mail is similar to telefax in that the data to be sent are digitized, transmitted via phone lines, and turned back into a document at the receiver. The advantage of electronic mail is in its compression factor. Whereas a telefax must scan a page in very fine lines and send all of the black and white information, electronic mail assigns characters fixed codes which can be transmitted as a few bits of information. Telefax presently has the advantage of being able to transmit an arbitrary page, including pictures, but electronic mail is beginning to deal with this problem. Electronic mail also integrates well with current directions in office automation, allowing documents prepared with sophisticated equipment at one site to be quickly transferred and printed at another site.

Currently, most electronic mail is intraorganizational, with mail transfer remaining within one computer. As computer networking becomes more common, however, it is becoming more feasible to communicate with anyone whose computer can be linked to your own via a network.

The pioneering efforts on general-purpose electronic mail were by organizations using the Defense Department's ARPANET.[1] The capability to send messages between computers existed before the ARPANET was developed, but it was used only in limited ways. With the advent of the ARPANET, tools began to be developed which made it convenient for individuals or organizations to distribute messages over broad geographic areas, using diverse computer facilities. The interest and activity in message systems has now reached such proportions that steps have been taken within the DoD to coordinate and unify the development of military message systems. The use of electronic mail is expected to increase dramatically in the next few years. The utility of such systems in the command and control and intelligence environments is clear, and applications in these areas will probably lead the way. As the costs for sending and handling electronic messags continue their rapid decrease, such uses can be expected to spread rapidly into other areas and, of course, will not be limited to the DoD.

A message system provides tools that help users (individuals or organizations) deal with messages in various ways. Messages must be composed, sent, received, stored, retrieved, forwarded, and replied to. Today's best interactive

computer systems provide a variety of word-processing and information handling capabilities. The message handling facilities should be well integrated with the rest of the system, so as to be a graceful extension of overall system capability.

The message system described in this report, MH, provides most of the features that can be found in other message systems and also incorporates some new ones. It has been built on the UNIX time-sharing system,[2] a popular operating system for the DEC PDP-11 and VAX classes of computers. A "secure" operating system similar to UNIX is currently being developed,[3] and that system will also run MH.

This report provides a complete description of MH and thus may serve as a user's manual, although parts of the report will be of interest to non-users as well. Sections 2 and 3, the Overview and Tutorial, present the key ideas of MH and will give those not familiar with message systems an idea of what such systems are like.

MH consists of a set of commands which use some special files and conventions. Section 4 covers the information a user needs to know in addition to the commands. The final section, Sec. 5, describes each of the MH commands in detail. A summary of the commands is given in Appendix A, and Appendixes B and C describe the ARPANET conventions for messages (we expect that many users of MH will be using the ARPANET) and the formal syntax of such messages, respectively. Finally, Appendix D provides an illustration of how MH commands may be used in conjunction with other UNIX facilities.

A novel approach has been taken in the design of MH. The design concept will be reported in detail in a forthcoming Rand report, but it can be described briefly as follows. Instead of creating a large subsystem that appears as a single command to the user, (such as MS[4]) MH is a collection of separate commands which are run as separate programs. The file and directory system of UNIX are used directly. Messages are stored as individual files (datasets), and collections of them are grouped into directories. In contrast, most other message systems store messages in a complicated data structure within a monolithic file. With the MH approach, UNIX commands can be interleaved with commands invoking the functions of the message handler. Conversely, existing UNIX commands can be used in connection with messages. For example, all the usual UNIX editing, text-formatting, and printing facilities can be applied directly to individual messages. MH, therefore, consists of a relatively small amount of new code; it makes extensive use of other UNIX software to provide the capabilities found in other message systems.

2. OVERVIEW

There are three main aspects of MH: the way messages are stored (the message database), the user's profile (which directs how certain actions of the message handler take place), and the commands for dealing with messages.

Under MH, each message is stored as a separate file. A user can take any action with a message that he could with an ordinary file in UNIX. A UNIX directory in which messages are stored is called a folder. Each folder contains some standard entries to support the message-handling functions. The messages in a folder have numerical names. These folders (directories) are entries in a particular directory path, described in the user profile, through which MH can find message folders. Using the UNIX "link" facility, it is possible for one copy of a message to be "filed" in more than one folder, providing a message index facility. Also, using the UNIX tree-structured file system, it is possible to have a folder within a folder. This two-level organization provides a "selection-list" facility, with the full power of the MH commands available on selected sublists of messages.

Each user of MH has a user profile, a file in his $HOME (initial login) directory called ". mh_profile". This profile contains several pieces of information used by the MH commands: a path name to the directory that contains the message folders, information concerning which folder the user last referenced (the "current" folder), and parameters that tailor MH commands to the individual user's requirements. It also contains most of the necessary state information concerning how the user is dealing with his messages, enabling MH to be implemented as a set of individual UNIX commands, in contrast to the usual approach of a monolithic subsystem.

In MH, incoming mail is appended to the end of a file called . mail in a user's $HOME directory. The user adds the new messages to his collection of MH messages by invoking the command *inc*. *Inc* (incorporate) adds the new messages to a folder called "inbox", assigning them names which are consecutive integers starting with the next highest integer available in inbox. *Inc* also produces a *scan* summary of the messages thus incorporated.

There are four commands for examining the messages in a folder: *show*, *prev*, *next*, and *scan*. *Show* displays a message in a folder, *prev* displays the message preceding the current message, and *next* displays the message following the current message. *Scan* summarizes the messages in a folder, producing one line per message, showing who the message is from, the date, the subject, etc.

The user may move a message from one folder to another with the command *file*. Messages may be removed from a folder by means of the command *rmm*. In addition, a user may query what the current folder is and may specify that a new folder become the current folder, through the command *folder*.

A set of messages based on content may be selected by use of the command *pick*. This command searches through messages in a folder and selects those that match a given criterion. A subfolder is created within the original folder, containing links to all the messages that satisfy the selection criteria.

A message folder (or subfolder) may be removed by means of the command *rmf*.

There are five commands enabling the user to create new messages and send them: *comp*, *dist*, *forw*, *repl*, and *send*. *Comp* provides the facility for the user to compose a new message; *dist* redistributes mail to additional addressees; *forw* enables the user to forward messages; and *repl* facilitates the generation of a reply to an incoming message. If a message is not sent directly by one of these commands, it may be sent at a later time using the command *send*.

All of the elements summarized above are described in more detail in the following sections. Many of the normal facilities of UNIX provide additional capabilities for dealing with messages in various ways. For example, it is possible to print messages on the line-printer without requiring any additional code within MH. Using standard UNIX facilities, any terminal output can be redirected to a file for repeated or future viewing. In general, the flexibility and capabilities of the UNIX interface with the user are preserved as a result of the integration of MH into the UNIX structure.

3. TUTORIAL

This tutorial provides a brief introduction to the MH commands. It should be sufficient to allow the user to read his mail, do some simple manipulations of it, and create and send messages.

A message has two major pieces: the header and the body. The body consists of the text of the message (whatever you care to type in). It follows the header and is separated from it by an empty line. (When you compose a message, the form that appears on your terminal shows a line of dashes after the header. This is for convenience and is replaced by an empty line when the message is sent.) The header is composed of several components, including the subject of the message and the person to whom it is addressed. Each component starts with a name and a colon; components must not start with a blank. The text of the component may take more than one line, but each continuation line must start with a blank. Messages typically have "to:", "cc:", and "subject:" components. When composing a message, you should include the "to:" and "subject:" components; the "cc:" (for people you want to send copies to) is not necessary.

The basic MH commands are *inc*, *scan*, *show*, *next*, *prev*, *rmm*, *comp*, and *repl*. These are described below.

inc

When you get the message "You have mail", type the command *inc*. You will get a "scan listing" such as:

```
7+    7/13   Cas        revival of measurement work
8    10/ 9   Norm       NBS people and publications
9    11/26   To:norm    question ≪Are there any functions
```

This shows the messages you received since the last time you executed this command (*inc* adds these new messages to your inbox folder). You can see this list again, plus a list of any other messages you have, by using the *scan* command.

scan

The scan listing shows the message number, followed by the date and the sender. (If you are the sender, the addressee in the "to:" component is displayed. You may send yourself a message by including your name among the "to:" or "cc:" addressees.) It also shows the message's subject; if the subject is short, the first part of the body of the message is included after the characters ≪.

show

This command shows the current message, that is, the first one of the new messages after an *inc*. If the message is not specified by name (number), it is generally the last message referred to by an MH command. For example,

show 5 will show message 5.

You can use the show command to copy a message or print a message.

show > x	will copy the message to file x.
show | print	will print the message, using the *print* command.
next	will show the message that follows the current message.
prev	will show the message previous to the current message.
rmm	will remove the current message.
rmm 3	will remove message 3.

comp

The *comp* command puts you in the editor to write or edit a message. Fill in or delete the "to:", "cc:", and "subject:" fields, as appropriate, and type the body of the message. Then exit normally from the editor. You will be asked "What now?". Type a carriage return to see the options. Typing **send** will cause the message to be sent; typing **quit** will cause an exit from *comp*, with the message draft saved.

If you quit without sending the message, it will be saved in a file called /usr/<name>/Mail/draft (where /usr/<name> is your $HOME directory). You can edit this file and send the message later, using the *send* command.

comp —editor prompter

This command uses a different editor and is useful for preparing "quick and dirty" messages. It prompts you for each component of the header. Type the information for that component, or type a carriage return to omit the component. After that, type the body of the message. Backspacing is the only form of editing allowed with this editor. When the body is complete, type a carriage return followed by <CTRL-D> (<OPEN> on Ann Arbor terminals). This completes the initial preparation of the message; from then on, use the same procedures as with *comp* (above).

repl
repl n

This command makes up an initial message form with a header that is appropriate for replying to an existing message. The message being answered is the current message if no message number is mentioned, or n if a number is specified. After the header is completed, you can finish the message as in *comp* (above).

This is enough information to get you going using MH. There are more commands, and the commands described here have more features. Subsequent sections explain MH in complete detail. The system is quite powerful if you want to use its sophisticated features, but the foregoing commands suffice for sending and receiving messages.

There are numerous additional capabilities you may wish to explore. For example, the *pick* command will select a subset of messages based on specified criteria such as sender or subject. Groups of messages may be designated, as described in Sec. V, under "Message Naming". The file ". mh_profile" can be used to tailor your use of the message system to your needs and preferences, as described in Sec. V, under "The User Profile". In general, you may learn additional features of the system selectively, according to your requirements, by studying the relevant sections of this manual. There is no need to learn all the details of the system at once.

4. DETAILED DESCRIPTION

This section describes the MH system in detail, including the components of the user profile, the conventions for message naming, and some of the other MH conventions. Readers who are generally familiar with computer systems will be able to follow the principal ideas, although some details may be meaningful only to those familiar with UNIX.

THE USER PROFILE

The first time an MH command is issued by a new user, the system prompts for a "path" and creates an MH "profile".

Each MH user has a profile which contains current state information for the MH package and, optionally, tailoring information for each individual program. When a folder becomes the current folder, it is recorded in the user's profile. Other profile entries control the MH path (where folders and special files are kept), folder and message protections, editor selection, and default arguments for each MH program.

The MH profile is stored in the file ". mh_profile" in the user's $HOME directory. It has the format of a message without any body. That is, each profile entry is on one line, with a keyword followed by a colon (:) followed by text particular to the keyword.

☛ *This file must not have blank lines.*

The keywords may have any combination of upper and lower case. (See Appendix B for a description of message formats.)

For the average MH user, the only profile entry of importance is "Path". Path specifies a directory in which MH folders and certain files such as "draft" are found. The argument to this keyword must be a legal UNIX path that names an existing directory. If this path is unrooted (i.e., does not begin with a /), it will be presumed to start from the user's $HOME directory. All folder and message references within MH will relate to this path unless full path names are used.

Message protection defaults to 664, and folder protection to 751. These may be changed by profile entries "Msg-Protect" and "Folder-Protect", respectively. The argument to these keywords is an octal number which is used as the UNIX file mode.[1]

When an MH program starts running, it looks through the user's profile for an entry with a keyword matching the program's name. For example, when *comp* is run, it looks for a "comp" profile entry. If one is found, the text of the profile entry is used as the default switch setting until all defaults are overridden by explicit switches passed to the program as arguments. Thus the profile entry "comp: —form standard.list" would direct *comp* to use the file "standard.list" as the message skeleton. If an explicit form switch is given to the *comp* command, it will override the switch obtained from the profile.

[1]See *chmod*(I) in the *UNIX Programmer's Manual*.[5]

In UNIX, a program may exist under several names, either by linking or aliasing. The actual invocation name is used by an MH program when scanning for its profile defaults. Thus, each MH program may have several names by which it can be invoked, and each name may have a different set of default switches. For example, if *comp* is invoked by the name *icomp*, the profile entry "icomp" will control the default switches for this invocation of the *comp* program. This provides a powerful definitional facility for commonly used switch settings.

The default editor for editing within *comp*, *repl*, *forw*, and *dist*, is "/bin/ned".[2] A different editor may be used by specifying the profile entry "Editor: ". The argument to "Editor" is the name of an executable program or shell command file which can be found via the user's $PATH defined search path, excluding the current directory. The "Editor:" profile specification may in turn be overridden by a "—editor <editor>" profile switch associated with *comp*, *repl*, *forw*, or *dist*. Finally, an explicit editor switch specified with any of these four commands will have ultimate precedence.

During message composition, more than one editor may be used. For example, one editor (such as *prompter*) may be used initially, and a second editor may be invoked later to revise the message being composed (see the discussion of *comp* in Section 5 for details). A profile entry "<lasteditor>—next: <editor>" specifies the name of the editor to be used after a particular editor. Thus "comp: —e prompter" causes the initial text to be collected by *prompter*, and the profile entry "prompter—next: ed" names ed as the editor to be invoked for the next round of editing.

Some of the MH commands, such as *show*, can be used on message folders owned by others, if those folders are readable. However, you cannot write in someone else's folder. All the MH command actions not requiring write permission may be used with a "read-only" folder. In a writable folder, a file named "cur" is used to contain its current message name. For read-only folders, the current message name is stored in the user's profile.

Table 1 lists examples of the currently defined profile entries, typical arguments, and the programs that reference the entries.

Table 1

PROFILE COMPONENTS

Keyword and Argument	MH Programs that Use Component
Path: Mail	All
Current-Folder: inbox	Most
Editor: /bin/ed	*comp*, *dist*, *forw*, *repl*
Msg—Protect: 644	*inc*
Folder—Protect: 711	*file*, *inc*, *pick*
<program>: default switches	All

[2]See Ref. 6 for a description of the NED text editor.

```
cur—<read-onlyfolder>: 172     Most
prompter—next: ed              comp, dist, forw, repl
```

Path should be present. Folder is maintained automatically by many MH commands (see the "Context" sections of the individual commands in Sec. V). All other entries are optional, defaulting to the values described above.

MESSAGE NAMING

Messages may be referred to explicitly or implicitly when using MH commands. A formal syntax of message names is given in Appendix C, but the following description should be sufficient for most MH users. Some details of message naming that apply only to certain commands are included in the description of those commands.

Most of the MH commands accept arguments specifying one or more folders, and one or more messages to operate on. The use of the word "msg" as an argument to a command means that exactly one message name may be specified. A message name may be a number, such as 1, 33, or 234, or it may be one of the "reserved" message names: first, last, prev, next, and cur. (As a shorthand, a period (.) is equivalent to cur.) The meanings of these names are straightforward: "first" is the first message in the folder; "last" is the last message in the folder; "prev" is the message numerically previous to the current message; "next" is the message numerically following the current message; "cur" (or ". ") is the current message in the folder.

The default in commands that take a "msg" argument is always "cur".

The word "msgs" indicates that several messages may be specified. Such a specification consists of several message designations separated by spaces. A message designation is either a message name or a message range. A message range is a specification of the form name1—name2 or name1:n, where name1 and name2 are message names and n is an integer. The first form designates all the messages from name1 to name2 inclusive; this must be a non-empty range. The second form specifies up to n messages, starting with name1 if name1 is a number, or first, cur, or next, and ending with name1 if name1 is last or prev. This interpretation of n is overridden if n is preceded by a plus sign or a minus sign; +n always means up to n messages starting with name1, and —n always means up to n messages ending with name1. Repeated specifications of the same message have the same effect as a single specification of the message. Examples of specifications are:

```
1 5 7—11 22
first 6 8 next
first—10
last:5
```

The message name "all" is a shorthand for "first—last", indicating all of the messages in the folder.

The limit on the number of messages in an expanded message list is generally 999—the maximum number of messages in a folder. However, the *show*

command and the commands *'pick —scan'* and *'pick —show'* are constrained to have argument lists that are no more than 512 characters long. (Under Version 7 UNIX this limit is 4096.)

In commands that accept "msgs" arguments, the default is either cur or all, depending on which makes more sense.

In all of the MH commands, a plus sign preceding an argument indicates a folder name. Thus, "+inbox" is the name of the user's standard inbox. If an explicit folder argument is given to an MH command, it will become the current folder (that is, the "Current-Folder:" entry in ". mh_profile" will be changed to this folder). In the case of the *file* and *pick* commands, which can have multiple output folders, a new source folder (other than the default current folder) is specified by "—src +folder".

OTHER MH CONVENTIONS

One very powerful feature of MH is that the MH commands may be issued from any current directory, and the proper path to the appropriate folder(s) will be taken from the user's profile. If the MH path is not appropriate for a specific folder or file, the automatic prepending of the MH path can be avoided by beginning a folder or file name with /. Thus any specific full path may be specified.

Arguments to the various programs may be given in any order, with the exception of a few switches whose arguments must follow immediately, such as "—src +folder" for *pick* and *file*.

Whenever an MH command prompts the user, the valid options will be listed in response to a <RETURN>. (The first of the listed options is the default if end-of-file is encountered, such as from a command file.) A valid response is any *unique* abbreviation of one of the listed options.

Standard UNIX documentation conventions are used in this report to describe MH command syntax. Arguments enclosed in brackets ([ ]) are optional; exactly one of the arguments enclosed within braces ({ }) must be specified, and all other arguments are required. The use of ellipsis dots (...) indicates zero or more repetitions of the previous item. For example, "+folder ..." would indicate that one or more "+folder" arguments is required and "[+folder ...]" indicates that 0 or more "+folder" arguments may be given.

MH departs from UNIX standards by using switches that consist of more than one character, e.g. "—header". To minimize typing, only a unique abbreviation of a switch need be typed; thus, for "—header", "—hea" is probably sufficient, depending on the other switches the command accepts. Each MH program accepts the switch "—help" (which *must* be spelled out fully) and produces a syntax description and a list of switches. In the list of switches, parentheses indicate required characters. For example, all "—help" switches will appear as "—(help)", indicating that no abbreviation is accepted.

Many MH switches have both on and off forms, such as "—format" and "—noformat". In many of the descriptions in Sec. V, only one form is defined; the other form, often used to nullify profile switch settings, is assumed to be the opposite.

MH COMMANDS

The MH package comprises 16 programs:

comp	Compose a message
dist	Redistribute a message
file	Move messages between folders
folder	Select/list status of folders
forw	Forward a message
inc	Incorporate new mail
next	Show the next message
pick	Select a set of messages by context
prev	Show the previous message
prompter	Prompting editor front end for composing messages
repl	Reply to a message
rmf	Remove a folder
rmm	Remove messages
scan	Produce a scan listing of selected messages
send	Send a previously composed message
show	Show messages

These programs are described below. The form of the descriptions conforms to the standard form for the description of UNIX commands.

NAME

comp — compose a message

SYNOPSIS

comp [—editor editor] [—form formfile] [file] [—use] [—nouse] [—help]

DESCRIPTION

Comp is used to create a new message to be mailed. If *file* is not specified, the file named "draft" in the user's MH directory will be used. *Comp* copies a message form to the file being composed and then invokes an editor on the file. The default editor is /bin/ned, which may be overridden with the '—editor' switch or with a profile entry "Editor:". (See Ref. 5 for a description of the NED text editing system.) The default message form contains the following elements:

To:
cc:
Subject:
————

If the file named "components" exists in the user's MH directory, it will be used instead of this form. If '—form formfile' is specified, the specified formfile (from the MH directory) will be used as the skeleton. The line of dashes or a blank line must be left between the header and the body of the message for the message to be identified properly when it is sent (see *send;*). The switch '—use' directs *comp* to continue editing an already started message. That is, if a *comp* (or *dist*, *repl*, or *forw*) is terminated without sending the message, the message can be edited again via "comp —use".

If the specified file (or draft) already exists, *comp* will ask if you want to delete it before continuing. A reply of **No** will abort the *comp*, **yes** will replace the existing draft with a blank skeleton, **list** will display the draft, and **use** will use it for further composition.

Upon exiting from the editor, *comp* will ask "What now?". The valid responses are **list**, to list the draft on the terminal; **quit**, to terminate the session and preserve the draft; **quit delete**, to terminate, then delete the draft; **send**, to send the message; **send verbose**, to cause the delivery process to be monitored; **edit <editor>**, to invoke <editor> for further editing; and **edit**, to re-edit using the same editor that was used on the preceding round unless a profile entry "<lasteditor>—next: <editor>" names an alternative editor.

Files

/etc/mh/components	The message skeleton
or <mh-dir>/components	Rather than the standard skeleton
$HOME/. mh_profile	The user profile
<mh-dir>/draft	The default message file
/usr/bin/send	To send the composed message

Profile Components

Path:	To determine the user's MH directory
Editor:	To override the use of /bin/ned as the default editor
<lasteditor>—next:	To name an editor to be used after exit from <lasteditor>

Defaults

'file' defaults to draft
'—editor' defaults to /bin/ned
'—nouse'

Context

Comp does not affect either the current folder or the current message.

NAME

dist — redistribute a message to additional addresses

SYNOPSIS

dist [+folder] [msg] [—form formfile] [—editor editor] [—annotate] [—noannotate] [—inplace] [—noinplace] [—help]

DESCRIPTION

Dist is similar to *forw*. It prepares the specified message for redistribution to addresses that (presumably) are not on the original address list. The file "distcomps" in the user's MH directory, or a standard form, or the file specified by '—form formfile' will be used as the blank components file to be prepended to the message being distributed. The standard form has the components "Distribute-to:" and "Distribute-cc:". When the message is sent, "Distribution-Date: date", "Distribution-From: name", and "Distribution-Id: id" (if '—msgid' is specified to *send*;) will be prepended to the outgoing message. Only those addresses in "Distribute-To", "Distribute-cc", and "Distribute-Bcc" will be sent. Also, a "Distribute-Fcc: folder" will be honored (see *send*;).

Send recognizes a message as a redistribution message by the existence of the field "Distribute-To:", so don't try to redistribute a message with only a "Distribute-cc:".

If the '—annotate' switch is given, each message being distributed will be annotated with the lines:

Distributed: ≪date≫
Distributed: Distribute-to: names

where each "to" list contains as many lines as required. This annotation will be done only if the message is sent directly from *dist*. If the message is not sent immediately from *dist* (i.e., if it is sent later via *send*;), "comp —use" may be used to re-edit and send the constructed message, but the annotations won't take place. The '—inplace' switch causes annotation to be done in place in order to preserve links to the annotated message.

See *comp* for a description of the '—editor' switch and for options upon exiting from the editor.

Files

/etc/mh/components	The message skeleton
or <mh-dir>/components	Rather than the standard skeleton
$HOME/.mh_profile	The user profile
<mh-dir>/draft	The default message file
/usr/bin/send	To send the composed message

Profile Components

Path:	To determine the user's MH directory
Editor:	To override the use of /bin/ned as the default editor
<lasteditor>—next:	To name an editor to be used after exit from <lasteditor>

Defaults

'+folder' defaults to the current folder
'msg' defaults to cur
'—editor' defaults to /bin/ned
'—noannotate'
'—noinplace'

Context

If a +folder is specified, it will become the current folder, and the current message will be set to the message being redistributed.

NAME

file — file message(s) in (an)other folder(s)

SYNOPSIS

file [—src +folder] [msgs] [—link] [—preserve] +folder ... [—nolink] [—nopreserve]
[—file file] [—nofile] [—help]

DESCRIPTION

File moves (*mv*(I)) or links (*ln*(I)) messages from a source folder into one or more destination folders. If you think of a message as a sheet of paper, this operation is not unlike filing the sheet of paper (or copies) in file cabinet folders. When a message is filed, it is linked into the destination folder(s) if possible, and is copied otherwise. As long as the destination folders are all on the same file system, multiple filing causes little storage overhead. This facility provides a good way to cross-file or multiply-index messages. For example, if a message is received from Jones about the ARPA Map Project, the command

file cur +jones +Map

would allow the message to be found in either of the two folders 'jones' or 'Map'.

The option '—file file' directs *file* to use the specified file as the source message to be filed, rather than a message from a folder.

If a destination folder doesn't exist, *file* will ask if you want to create one. A negative response will abort the file operation.

'—link' preserves the source folder copy of the message (i.e., it does a *ln*(I) rather than a *mv*(I)), whereas, '—nolink' deletes the "filed" messages from the source folder. Normally, when a message is filed, it is assigned the next highest number available in each of the destination folders. Use of the '—preserve' switch will override this message "renaming", but name conflicts may occur, so use this switch cautiously. (See *pick* for more details on message numbering.)

If '—link' is not specified (or '—nolink' is specified), the filed messages will be removed (unlink(II)) from the source folder.

Files

$HOME/. mh_profile	The user profile

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder
Folder—Protect:	To set mode when creating a new folder

Defaults

'—src +folder' defaults to the current folder
'msgs' defaults to cur
'—nolink'
'—nopreserve'
'—nofile'

Context

If '—src +folder' is given, it will become the current folder for future MH commands. If neither '—link' nor 'all' are specified, the current message in the source folder will be set to the last message specified; otherwise, the current message won't be changed.

NAME

folder — set/list current folder/message

SYNOPSIS

folder [+folder] [msg] [—all] [—fast] [—nofast] [—up] [—down] [—header] [—noheader] [—total] [—nototal] [—pack] [—nopack] [—help]

folders <equivalent to 'folder —all'>

DESCRIPTION

Since the MH environment is the shell, it is easy to lose track of the current folder from day to day. *Folder* will list the current folder, the number of messages in it, the range of the messages (low-high), and the current message within the folder, and will flag a selection list or extra files if they exist. An example of the output is:

inbox+ has 16 messages (3— 22); cur= 5.

If a '+folder' and/or 'msg' are specified, they will become the current folder and/or message. An '—all' switch will produce a line for each folder in the user's MH directory, sorted alphabetically. These folders are preceded by the read-only folders, which occur as . mh_profile "cur—" entries. For example,

```
                Folder      # of messages (  range  ); cur msg (other files)
     /fsd/rs/m/tacc  has   35 messages (  1—  35); cur= 23.
 /rnd/phyl/Mail/EP  has   82 messages (  1—108); cur= 82.
                ff  has    4 messages (  1—   4); cur=  1.
            inbox+  has   16 messages (  3—  22); cur=  5.
                mh  has   76 messages (  1—  76); cur= 70.
             notes  has    2 messages (  1—   2); cur=  1.
              ucom  has  124 messages (  1—124); cur=  6; (select).

              TOTAL= 339 messages in   7 Folders.
```

The "+" after inbox indicates that it is the current folder. The "(select)" indicates that the folder ucom has a selection list produced by *pick*. If "others" had appeared in parentheses at the right of a line, it would indicate that there are files in the folder directory that don't belong under the MH file naming scheme.

The header is output if either an '—all' or a '—header' switch is specified; it is suppressed by '—noheader'. Also, if *folder* is invoked by a name ending with "s" (e.g., *folders*), '—all' is assumed. A '—total' switch will produce only the summary line.

If '—fast' is given, only the folder name (or names in the case of '—all') will be listed. (This is faster because the folders need not be read.)

The switches '—up' and '—down' change the folder to be the one above or below the current folder. That is, "folder —down" will set the folder to "<current—folder>/select", and if the current folder is a selection-list folder, "folder —up" will set the current folder to the parent of the selection-list. (See *pick* for details on selection-lists.)

The '—pack' switch will compress the message names in a folder, removing holes in message

numbering.

Files

$HOME/.mh_profile	The user profile
/bin/ls	To fast-list the folders

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

Defaults

'+folder' defaults to the current folder
'msg' defaults to none
'—nofast'
'—noheader'
'—nototal'
'—nopack'

Context

If '+folder' and/or 'msg' are given, they will become the current folder and/or message.

NAME

forw — forward messages

SYNOPSIS

forw [+folder] [msgs] [—editor editor] [—form formfile] [—annotate] [—noannotate] [—inplace] [—noinplace] [—help]

DESCRIPTION

Forw may be used to prepare a message containing other messages. It constructs the new message from the components file or '—form formfile' (see *comp*), with a body composed of the message(s) to be forwarded. An editor is invoked as in *comp*, and after editing is complete, the user is prompted before the message is sent.

If the '—annotate' switch is given, each message being forwarded will be annotated with the lines

Forwarded: ≪date≫
Forwarded: To: names
Forwarded: cc: names

where each "To:" and "cc:" list contains as many lines as required. This annotation will be done only if the message is sent directly from *forw*. If the message is not sent immediately from *forw*, "comp —use" may be used in a later session to re-edit and send the constructed message, but the annotations won't take place. The '—inplace' switch permits annotating a message in place in order to preserve its links.

See *comp* for a description of the '—editor' switch.

Files

/etc/mh/components	The message skeleton
or <mh-dir>/components	Rather than the standard skeleton
$HOME/. mh_profile	The user profile
<mh-dir>/draft	The default message file
/usr/bin/send	To send the composed message

Profile Components

Path:	To determine the user's MH directory
Editor:	To override the use of /bin/ned as the default editor
Current-Folder:	To find the default current folder
<lasteditor>—next:	To name an editor to be used after exit from <lasteditor>

Defaults

'+folder' defaults to the current folder
'msgs' defaults to cur
'—editor' defaults to /bin/ned
'—noannotate'
'—noinplace'

Context

If a +folder is specified, it will become the current folder, and the current message will be set to the first message being forwarded.

NAME

inc — incorporate new mail

SYNOPSIS

inc [+folder] [—audit audit-file] [—help]

DESCRIPTION

Inc incorporates mail from the user's incoming mail drop (. mail) into an MH folder. If '+folder' isn't specified, the folder named "inbox" in the user's MH directory will be used. The new messages being incorporated are assigned numbers starting with the next highest number in the folder. If the specified (or default) folder doesn't exist, the user will be queried prior to its creation. As the messages are processed, a *scan* listing of the new mail is produced.

If the user's profile contains a "Msg—Protect: nnn" entry, it will be used as the protection on the newly created messages, otherwise the MH default of 664 will be used. During all operations on messages, this initially assigned protection will be preserved for each message, so *chmod*(I) may be used to set a protection on an individual message, and its protection will be preserved thereafter.

If the switch '—audit audit-file' is specified (usually as a default switch in the profile), then *inc* will append a header line and a line per message to the end of the specified audit-file with the format:

≪inc≫ date
<scan line for first message>
<scan line for second message>
<etc.>

This is useful for keeping track of volume and source of incoming mail. Eventually, *repl*, *forw*, *comp*, and *dist* may also produce audits to this (or another) file, perhaps with "Message-Id:" information to keep an exact correspondence history. "Audit-file" will be in the user's MH directory unless a full path is specified.

Inc will incorporate even illegally formatted messages into the user's MH folder, inserting a blank line prior to the offending component and printing a comment identifying the bad message.

In all cases, the . mail file will be zeroed.

Files

$HOME/. mh_profile	The user profile
$HOME/. mail	The user's mail drop
<mh-dir>/audit-file	Audit trace file (optional)

Profile Components

Path:	To determine the user's MH directory
Folder—Protect:	For protection on new folders
Msg—Protect:	For protection on new messages

Defaults

'+folder' defaults to "inbox"

Context

The folder into which the message is being incorporated will become the current folder, and the first message incorporated will be the current message. This leaves the context ready for a *show* of the first new message.

NAME

next — show the next message

SYNOPSIS

next [+folder] [—switches for *l*] [—help]

DESCRIPTION

Next performs a *show* on the next message in the specified (or current) folder. Like *show*, it passes any switches on to the program *l*, which is called to list the message. This command is exactly equivalent to "show next".

Files

$HOME/. mh_profile	The user profile

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

Defaults

Context

If a folder is specified, it will become the current folder, and the message that is shown (i.e., the next message in sequence) will become the current message.

NAME

pick — select messages by content

SYNOPSIS

pick { —cc | —date | —from | —search | —subject | —to | ——component } [—src +folder] [msgs] [—help] [—scan] [—noscan] [—show] [—noshow] [—nofile] [—nokeep] pattern [—file [—preserve] [—link] +folder ... [—nopreserve] [—nolink]] [—keep [—stay] [—nostay] [+folder ...]]

typically:

pick —from jones —scan
pick —to holloway
pick —subject ned —scan —keep

DESCRIPTION

Pick searches messages within a folder for the specified contents, then performs several operations on the selected messages.

A modified *grep*(I) is used to perform the searching, so the full regular expression (see *ed*(I)) facility is available within 'pattern'. With '—search', pattern is used directly, and with the others, the grep pattern constructed is:

"component:.*pattern"

This means that the pattern specified for a '—search' will be found everywhere in the message, including the header and the body, while the other search requests are limited to the single specified component. The expression '——component pattern' is a shorthand for specifying '—search "component:.*pattern" '; it is used to pick a component not in the set [cc date from subject to]. An example is "pick ——reply—to pooh —show".

Searching is performed on a per-line basis. Within the header of the message, each component is treated as one long line, but in the body, each line is separate. Lower-case letters in the search pattern will match either lower or upper case in the message, while upper case will match only upper case.

Once the search has been performed, the selected messages are scanned (see *scan*) if the '—scan' switch is given, and then they are shown (see *show*) if the '—show' switch is given. After these two operations, the file operations (if requested) are performed.

The '—file' switch operates exactly like the *file* command, with the same meaning for the '—preserve' and '—link' switches.

The '—keep' switch is similar to '—file', but it produces a folder that is a subfolder of the folder being searched and defines it as the current folder (unless the '—stay' flag is used). This subfolder contains the messages which matched the search criteria. All of the MH commands may be used with the sub-folder as the current folder. This gives the user considerable power in dealing with subsets of messages in a folder.

The messages in a folder produced by '—keep' will always have the same numbers as they have in the source folder (i.e., the '—preserve' switch is automatic). This way, the message numbers are consistent with the folder from which the messages were selected. Messages are not removed from the source folder (i.e., the '—link' switch is assumed). If a '+folder' is not specified, the standard name "select" will be used. (This is the meaning of "(select)" when it appears in the output of the *folder* command.) If '+folder' arguments are given to '—keep', they will be used rather than "select" for the names of the subfolders. This allows for several subfolders to be maintained concurrently.

When a '—keep' is performed, the subfolder becomes the current folder. This can be overridden by use of the '—stay' switch.

Here's an example:

```
 1 % folder +inbox
 2        inbox+ has  16 messages (  3— 22); cur=  3.
 3 % pick —from dcrocker
 4 6 hits.
 5 [+inbox/select now current]
 6 % folder
 7  inbox/select+ has   6 messages (  3— 16); cur=  3.
 8 % scan
 9   3+ 6/20  Dcrocker        Re: ned file update issue...
10   6  6/23  Dcrocker        removal of files from /tm...
11   8  6/27  Dcrocker        Problems with the new ned...
12  13  6/28  dcrocker        newest nned  <<I would ap...
13  15  7/ 5  Dcrocker        nned  <<Last week I asked...
14  16  7/ 5  dcrocker        message id format  <<I re...
15 % show all | print
16   [produce a full listing of this set of messages on the line printer.]
17 % folder —up
18        inbox+ has  16 messages (  3— 22); cur=  3; (select).
19 % folder —down
20  inbox/select+ has  6 messages (  3— 16); cur=  3.
21 % rmf
22 [+inbox now current]
23 % folder
24        inbox+ has  16 messages (  3— 22); cur=  3.
```

This is a rather lengthy example, but it shows the power of the MH package. In item 1, the current folder is set to inbox. In 3, all of the messages from dcrocker are found in inbox and linked into the folder "inbox/select". (Since no action switch is specified, '—keep' is assumed.) Items 6 and 7 show that this subfolder is now the current folder. Items 8 through 14 are a *scan* of the selected messages (note that they are all from dcrocker and are all in upper and lower case). Item 15 lists all of the messages to the high-speed printer. Item 17 directs *folder* to set the current folder to the parent of the selection-list folder, which is now current. Item 18 shows that this has been done. Item 19 resets the current folder to the selection list, and 21 removes the selection-list folder and resets the current folder to the parent folder, as shown in 22 and 23.

Files

$HOME/.mh_profile	The user profile

Profile Components

Path:	To determine the user's MH directory
Folder—Protect:	For protection on new folders
Current-Folder:	To find the default current folder

Defaults

'—src +folder' defaults to current
'msgs' defaults to all
'—keep +select' is the default if no '—scan', '—show', or '—file' is specified

Context

If a '—src +folder' is specified, it will become the current folder, unless a '—keep' with 0 or 1 folder arguments makes the selection-list subfolder the current folder. Each selection-list folder will have its current message set to the first of the messages linked into it unless the selection list already existed, in which case the current message won't be changed.

NAME

prev — show the previous message

SYNOPSIS

prev [+folder] [—switches for *l*] [—help]

DESCRIPTION

Prev performs a *show* on the previous message in the specified (or current) folder. Like *show*, it passes any switches on to the program *l*, which is called to list the message. This command is exactly equivalent to "show prev".

Files

$HOME/.mh_profile	The user profile

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

Defaults

Context

If a folder is specified, it will become current, and the message that is shown (i.e., the previous message in sequence) will become the current message.

NAME

prompter — prompting editor front end

SYNOPSIS

This program is not called directly but takes the place of an editor and acts as an editor front end.

prompter [—erase chr] [—kill chr] [—help]

DESCRIPTION

Prompter is an editor which allows rapid composition of messages. It is particularly useful to network and low-speed (less than 2400 baud) users of MH. It is an MH program in that it can have its own profile entry with switches, but it can't be invoked directly as all other MH commands can; it is an editor in that it is invoked by an "—editor prompter" switch or by the profile entry "Editor: prompter", but functionally it is merely a text-collector and not a true editor.

Prompter expects to be called from *comp*, *repl*, *dist*, or *forw*, with a draft file as an argument. For example, "comp —editor prompter" will call *prompter* with the file "draft" already set up with blank components. For each blank component it finds in the draft, it prompts the user and accepts a response. A <RETURN> will cause the whole component to be left out. A "\" preceding a <RETURN> will continue the response on the next line, allowing for multiline components.

Any component that is non-blank will be copied and echoed to the terminal.

The start of the message body is prompted by a line of dashes. If the body is non-blank, the prompt is "———Enter additional text". Message-body typing is terminated with a <CTRL-D> (or <OPEN>). Control is returned to the calling program, where the user is asked "What now?". See *comp* for the valid options.

The line editing characters for kill and erase may be specified by the user via the arguments "—kill chr" and "—erase chr", where chr may be a character; or "\nnn", where nnn is the octal value for the character. (Again, these may come from the default switches specified in the user's profile.)

A <DEL> during message-body typing is equivalent to <CTRL-D> for compatibility with NED. A <DEL> during component typing will abort the command that invoked *prompter*.

Files

None

Profile Components

prompter-next:	To name the editor to be used on exit from *prompter*

Defaults

Context

None

NAME

repl — reply to a message

SYNOPSIS

repl [+folder] [msg] [—editor editor] [—inplace] [—annotate] [—help] [—noinplace] [—noannotate]

DESCRIPTION

Repl aids a user in producing a reply to an existing message. In its simplest form (with no arguments), it will set up a message-form skeleton in reply to the current message in the current folder, invoke the editor, and send the composed message if so directed. The composed message is constructed as follows:

To: <Reply-To> or <From>
cc: <cc>, <To>
Subject: Re: <Subject>
In-reply-to: Your message of <Date>
<Message-Id>

where field names enclosed in angle brackets (< >) indicate the contents of the named field from the message to which the reply is being made. Once the skeleton is constructed, an editor is invoked (as in *comp*, *dist*, and *forw*). While in the editor, the message being replied to is available through a link named "@". In NED, this means the replied-to message may be "used" with "use @", or put in a window by "window @".

As in *comp*, *dist*, and *forw*, the user will be queried before the message is sent. If '—annotate' is specified, the replied-to message will be annotated with the single line

Replied: ≪Date≫.

The command "comp —use" may be used to pick up interrupted editing, as in *dist* and *forw*; the '—inplace' switch annotates the message in place, so that all folders with links to it will see the annotation.

Files

$HOME/. mh_profile	The user profile
<mh-dir>/draft	The constructed message file
/usr/bin/send	To send the composed message

Profile Components

Path:	To determine the user's MH directory
Editor:	To override the use of /bin/ned as the default editor
Current-Folder:	To find the default current folder

Defaults

'+folder' defaults to current
'msgs' defaults to cur
'—editor' defaults to /bin/ned
'—noannotate'
'—noinplace'

Context

If a '+folder' is specified, it will become the current folder, and the current message will be set to the replied-to message.

NAME

rmf — remove folder

SYNOPSIS

rmf [+folder] [—help]

DESCRIPTION

Rmf removes all of the files (messages) within the specified (or default) folder, and then removes the directory (folder). If there are any files within the folder which are not a part of MH, they will *not* be removed, and an error will be produced. If the folder is given explicitly or the current folder is a subfolder (i.e., a selection list from *pick*), it will be removed without confirmation. If no argument is specified and the current folder is not a selection-list folder, the user will be asked for confirmation.

Rmf irreversibly deletes messages that don't have other links, so use it with caution.

If the folder being removed is a subfolder, the parent folder will become the new current folder, and *rmf* will produce a message telling the user this has happened. This provides an easy mechanism for selecting a set of messages, operating on the list, then removing the list and returning to the current folder from which the list was extracted. (See the example under *pick*.)

The files that *rmf* will delete are cur, any file beginning with a comma, and files with purely numeric names. All others will produce error messages.

Rmf of a read-only folder will delete the "cur—" entry from the profile without affecting the folder itself.

Files

$HOME/. mh_profile	The user profile

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

Defaults

'+folder' defaults to current, usually with confirmation

Context

Rmf will set the current folder to the parent folder if a subfolder is removed; or if the current folder is removed, it will make "inbox" current. Otherwise, it doesn't change the current folder or message.

NAME

rmm — remove messages

SYNOPSIS

rmm [+folder] [msgs] [—help]

DESCRIPTION

Rmm removes the specified messages by renaming the message files with preceding commas. (This is the Rand-UNIX backup file convention.)

The current message is not changed by *rmm*, so a *next* will advance to the next message in the folder as expected.

Files

$HOME/. mh_profile	The user profile

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

Defaults

'+folder' defaults to current
'msgs' defaults to cur

Context

If a folder is given, it will become current.

NAME

scan — produce a one-line-per-message scan listing

SYNOPSIS

scan [+folder] [msgs] [—ff] [—header] [—help] [—noff] [—noheader]

DESCRIPTION

Scan produces a one-line-per-message listing of the specified messages. Each *scan* line contains the message number (name), the date, the "From" field, the "Subject" field, and, if room allows, some of the body of the message. For example:

```
 #      Date      From          Subject  [≪Body]
15+     7/ 5      Dcrocker      nned  ≪Last week I asked some of
16 —    7/ 5      dcrocker      message id format  ≪I recommend
18      7/ 6      Obrien        Re: Exit status from mkdir
19      7/ 7      Obrien        "scan" listing format in MH
```

The '+' on message 15 indicates that it is the current message. The '—' on message 16 indicates that it has been replied to, as indicated by a "Replied:" component produced by an '—annotate' switch to the *repl* command.

If there is sufficient room left on the *scan* line after the subject, the line will be filled with text from the body, preceded by ≪. *Scan* actually reads each of the specified messages and parses them to extract the desired fields. During parsing, appropriate error messages will be produced if there are format errors in any of the messages.

The '—header' switch produces a header line prior to the *scan* listing, and the '—ff' switch will cause a form feed to be output at the end of the *scan* listing. See Appendix D.

Files

$HOME/. mh_profile	The user profile

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

Defaults

Defaults:
'+folder' defaults to current
'msgs' defaults to all
'—noff'
'—noheader'

Context

If a folder is given, it will become current. The current message is unaffected.

NAME

send — send a message

SYNOPSIS

send [file] [—draft] [—verbose] [—format] [—msgid] [—help] [—noverbose] [—noformat] [—nomsgid]

DESCRIPTION

Send will cause the specified file (default <mh-dir>/draft) to be delivered to each of the addresses in the "To:", "cc:", and "Bcc:" fields of the message. If '—verbose' is specified, *send;* will monitor the delivery of local and net mail. *Send* with no argument will query whether the draft is the intended file, whereas '—draft' will suppress this question. Once the message has been mailed (or queued) successfully, the file will be renamed with a leading comma, which allows it to be retreived until the next draft message is sent. If there are errors in the formatting of the message, *send;* will abort with a (hopefully) helpful error message.

If a "Bcc:" field is encountered, its addresses will be used for delivery, but the "Bcc:" field itself will be deleted from all copies of the outgoing message.

Prior to sending the message, the fields "From: user", and "Date: now" will be prepended to the message. If '—msgid' is specified, then a "Message-Id:" field will also be added to the message. If the message already contains a "From:" field, then a "Sender: user" field will be added instead. (An already existing "Sender:" field will be deleted from the message.)

If the user doesn't specify '—noformat', each of the entries in the "To:" and "cc:" fields will be replaced with "standard" format entries. This standard format is designed to be usable by all of the message handlers on the various systems around the ARPANET.

If an "Fcc: folder" is encountered, the message will be copied to the specified folder in the format in which it will appear to any receivers of the message. That is, it will have the prepended fields and field reformatting.

If a "Distribute-To:" field is encountered, the message is handled as a redistribution message (see *dist* for details), with "Distribution-Date: now" and "Distribution-From: user" added.

Files

$HOME/. mh_profile	The user profile

Profile Components

Path:	To determine the user's MH directory

Defaults

'file' defaults to draft
'—noverbose'
'—format'
'—nomsgid'

Context

Send has no effect on the current message or folder.

NAME

show — show (list) messages

SYNOPSIS

show [+folder] [msgs] [—pr] [—nopr] [—draft] [—help] [*l* or *pr* switches]

DESCRIPTION

Show lists each of the specified messages to the standard output (typically, the terminal). The messages are listed exactly as they are, with no reformatting. A program called *l* is invoked to do the listing, and any switches not recognized by *show* are passed along to *l*.

If no "msgs" are specified, the current message is used. If more than one message is specified, *l* will prompt for a <return> prior to listing each message.

l will list each message, a page at a time. When the end of page is reached, *l* will ring the bell and wait for a <RETURN> or <CTRL-D>. If a <return> is entered, *l* will clear the screen before listing the next page, whereas <CTRL-D> will not. The switches to *l* are '—p#' to indicate the page length in lines, and '—w#' to indicate the width of the page in characters.

If the standard output is not a terminal, no queries are made, and each file is listed with a one-line header and two lines of separation.

If '—pr' is specified, then *pr*(I) will be invoked rather than *l*, and the switches (other than '—draft') will be passed along. "Show —draft" will list the file <mh-dir>/draft if it exists.

Files

$HOME/. mh_profile	The user profile
/bin/l	Screen-at-a-time list program
/bin/pr	*pr*(I)

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

Defaults

'+folder' defaults to current
'msgs' defaults to cur
'—nopr'

Context

If a folder is given, it will become the current message. The last message listed will become the current message.

Appendix A

COMMAND SUMMARY[3]

comp [—editor editor] [—form formfile] [file] [—use] [—nouse] [—help]

dist [+folder] [msg] [—form formfile] [—editor editor] [—annotate] [—noannotate] [—inplace] [—noinplace] [—help]

file [—src +folder] [msgs] [—link] [—preserve] +folder ... [—nolink] [—nopreserve] [—file file] [—nofile] [—help]

folder [+folder] [msg] [—all] [—fast] [—nofast] [—up] [—down] [—header] [—noheader] [—total] [—nototal] [—pack] [—nopack] [—help]

forw [+folder] [msgs] [—editor editor] [—form formfile] [—annotate] [—noannotate] [—inplace] [—noinplace] [—help]

inc [+folder] [—audit audit-file] [—help]

next [+folder] [—switches for *l*] [—help]

pick { —cc, —date, —from, —search, —subject, —to, ——component } [—src +folder] [msgs] [—help] [—scan] [—noscan] [—show] [—noshow] [—nofile] [—nokeep] pattern [—file [—preserve] [—link] +folder ... [—nopreserve] [—nolink]] [—keep [—stay] [—nostay] [+folder ...]]

prev [+folder] [—switches for *l*] [—help]

prompter [—erase chr] [—kill chr] [—help]

repl [+folder] [msg] [—editor editor] [—inplace] [—annotate] [—help] [—noinplace] [—noannotate]

rmf [+folder] [—help]

rmm [+folder] [msgs] [—help]

scan [+folder] [msgs] [—ff] [—header] [—help] [—noff] [—noheader]

send [file] [—draft] [—verbose] [—format] [—msgid] [—help] [—noverbose] [—noformat] [—nomsgid]

show [+folder] [msgs] [—pr] [—nopr] [—draft] [—help] [*l* or *pr* switches]

[3]All commands accept a —help switch.

Appendix B

MESSAGE FORMAT

This section paraphrases the format of ARPANET text messages given in Ref. 6.

ASSUMPTIONS

(1) Messages are expected to consist of lines of text. Graphics and binary data are not handled.

(2) No data compression is accepted. All text is clear ASCII 7-bit data.

LAYOUT

A general "memo" framework is used. A message consists of a block of information in a rigid format, followed by general text with no specified format. The rigidly formatted first part of a message is called the header, and the free-format portion is called the body. The header must always exist, but the body is optional.

THE HEADER

Each header item can be viewed as a single logical line of ASCII characters. If the text of a header item extends across several real lines, the continuation lines are indicated by leading spaces or tabs.

Each header item is called a component and is composed of a keyword or name, along with associated text. The keyword begins at the left margin, may contain spaces or tabs, may not exceed 63 characters, and is terminated by a colon (:). Certain components (as identified by their keywords) must follow rigidly defined formats in their text portions.

The text for most formatted components (e.g., "Date:" and "Message-Id:") is produced automatically. The only ones entered by the user are address fields such as "To:", "cc:", etc. ARPA addresses are assigned mailbox names and host computer specifications. The rough format is "mailbox at host", such as "Borden at Rand-Unix". Multiple addresses are separated by commas. A missing host is assumed to be the local host.

THE BODY

A blank line signals that all following text up to the end of the file is the body. (A blank line is defined as a pair of <end-of-line> characters with *no* characters in between.) No formatting is expected or enforced within the body.

Within MH, a line consisting of dashes is accepted as the header delimiter. This is a cosmetic feature applying only to locally composed mail.

Appendix C
MESSAGE NAME BNF

```
msgs          :=   msgspec                |
                   msgs msgspec

msgspec       :=   msg                    |
                   msg-range               |
                   msg-sequence

msg           :=   msg-name               |
                   <number>

msg-name      :=   "first"                |
                   "last"                  |
                   "cur"                   |
                   "."                     |
                   "next"                  |
                   "prev"

msg-range     :=   msg"-"msg              |
                   "all"

msg-sequence  :=   msg":"signed-number

signed-number :=   "+"<number>            |
                   "—"<number>             |
                   <number>
```

Where <number> is a decimal number in the range 1 to 999.

Msg-range specifies all of the messages in the given range and must not be empty.

Msg-sequence specifies up to <number> of messages, beginning with "msg" (in the case of first, cur, next, or <number>), or ending with "msg" (in the case of prev or last). +<number> forces "starting with msg", and —<number> forces "ending with number". In all cases, "msg" must exist.

Appendix D

EXAMPLE OF SHELL COMMANDS

UNIX commands may be mixed with MH commands to obtain additional functions. These may be prepared as files (known as shell command files or shell scripts). The following example is a useful function that illustrate the possibilities. Other functions, such as copying, deleting, renaming, etc., can be achieved in a similar fashion.

HARDCOPY

The command:

```
(scan —ff —header; show all —pr —f) | print
```

produces a scan listing of the current folder, followed by a form feed, followed by a formatted listing of all messages in the folder, one per page. Omitting "—pr —f" will cause the messages to be concatenated, separated by a one-line header and two blank lines.

You can create variations on this theme, using *pick*.

REFERENCES

1. Crocker, D. H., J. J. Vittal, K. T. Pogran, and D. A. Henderson, Jr., "Standard for the Format of ARPA Network Test Messages," *Arpanet Request for Comments*, No. 733, Network Information Center 41952, Augmentation Research Center, Stanford Research Institute, November 1977.

2. Thompson, K., and D. M. Ritchie, "The UNIX Time-sharing System," *Communications of the ACM*, Vol. 17, July 1974, pp. 365-375.

3. McCauley, E. J., and P. J. Drongowski, "KSOS—The Design of a Secure Operating System," *AFIPS Conference Proceedings*, National Computer Conference, Vol. 48, 1979, pp. 345-353.

4. Crocker, David H., *Framework and Functions of the "MS" Personal Message System*, The Rand Corporation, R-2134-ARPA, December 1977.

5. Thompson, K., and D. M. Ritchie, *UNIX Programmer's Manual*, 6th ed., Western Electric Company, May 1975 (available only to UNIX licensees).

6. Bilofsky, Walter, *The CRT Text Editor NED—Introduction and Reference Manual*, The Rand Corporation, R-2176-ARPA, December 1977.

UNIX† Assembler Reference Manual

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

0. Introduction

This document describes the usage and input syntax of the UNIX PDP-11 assembler *as*. The details of the PDP-11 are not described.

The input syntax of the UNIX assembler is generally similar to that of the DEC assembler PAL-11R, although its internal workings and output format are unrelated. It may be useful to read the publication DEC-11-ASDB-D, which describes PAL-11R, although naturally one must use care in assuming that its rules apply to *as*.

As is a rather ordinary assembler without macro capabilities. It produces an output file that contains relocation information and a complete symbol table; thus the output is acceptable to the UNIX link-editor *ld*, which may be used to combine the outputs of several assembler runs and to obtain object programs from libraries. The output format has been designed so that if a program contains no unresolved references to external symbols, it is executable without further processing.

1. Usage

as is used as follows:

as [**−u**] [**−o** *output*] *file*$_1$ ...

If the optional "−u" argument is given, all undefined symbols in the current assembly will be made undefined-external. See the **.globl** directive below.

The other arguments name files which are concatenated and assembled. Thus programs may be written in several pieces and assembled together.

The output of the assembler is by default placed on the file *a.out* in the current directory; the "−o" flag causes the output to be placed on the named file. If there were no unresolved external references, and no errors detected, the output file is marked executable; otherwise, if it is produced at all, it is made non-executable.

2. Lexical conventions

Assembler tokens include identifiers (alternatively, "symbols" or "names"), temporary symbols, constants, and operators.

2.1 Identifiers

An identifier consists of a sequence of alphanumeric characters (including period ".", underscore "_", and tilde "~" as alphanumeric) of which the first may not be numeric. Only the first eight characters are significant. When a name begins with a tilde, the tilde is discarded and that occurrence of the identifier generates a unique entry in the symbol table which can match no other occurrence of the identifier. This feature is used by the C compiler to place

† UNIX is a Trademark of Bell Laboratories.

names of local variables in the output symbol table without having to worry about making them unique.

2.2 Temporary symbols

A temporary symbol consists of a digit followed by "f" or "b". Temporary symbols are discussed fully in §5.1.

2.3 Constants

An octal constant consists of a sequence of digits; "8" and "9" are taken to have octal value 10 and 11. The constant is truncated to 16 bits and interpreted in two's complement notation.

A decimal constant consists of a sequence of digits terminated by a decimal point ".". The magnitude of the constant should be representable in 15 bits; i.e., be less than 32,768.

A single-character constant consists of a single quote "'" followed by an ASCII character not a new-line. Certain dual-character escape sequences are acceptable in place of the ASCII character to represent new-line and other non-graphics (see *String statements*, §5.5). The constant's value has the code for the given character in the least significant byte of the word and is null-padded on the left.

A double-character constant consists of a double quote """ followed by a pair of ASCII characters not including new-line. Certain dual-character escape sequences are acceptable in place of either of the ASCII characters to represent new-line and other non-graphics (see *String statements*, §5.5). The constant's value has the code for the first given character in the least significant byte and that for the second character in the most significant byte.

2.4 Operators

There are several single- and double-character operators; see §6.

2.5 Blanks

Blank and tab characters may be interspersed freely between tokens, but may not be used within tokens (except character constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

2.6 Comments

The character "/" introduces a comment, which extends through the end of the line on which it appears. Comments are ignored by the assembler.

3. Segments

Assembled code and data fall into three segments: the text segment, the data segment, and the bss segment. The text segment is the one in which the assembler begins, and it is the one into which instructions are typically placed. The UNIX system will, if desired, enforce the purity of the text segment of programs by trapping write operations into it. Object programs produced by the assembler must be processed by the link-editor *ld* (using its "−n" flag) if the text segment is to be write-protected. A single copy of the text segment is shared among all processes executing such a program.

The data segment is available for placing data or instructions which will be modified during execution. Anything which may go in the text segment may be put into the data segment. In programs with write-protected, sharable text segments, data segment contains the initialized but variable parts of a program. If the text segment is not pure, the data segment begins immediately after the text segment; if the text segment is pure, the data segment begins at the lowest 8K byte boundary after the text segment.

The bss segment may not contain any explicitly initialized code or data. The length of the

bss segment (like that of text or data) is determined by the high-water mark of the location counter within it. The bss segment is actually an extension of the data segment and begins immediately after it. At the start of execution of a program, the bss segment is set to 0. Typically the bss segment is set up by statements exemplified by

```
lab: . = .+10
```

The advantage in using the bss segment for storage that starts off empty is that the initialization information need not be stored in the output file. See also *Location counter* and *Assignment statements* below.

4. The location counter

One special symbol, "." , is the location counter. Its value at any time is the offset within the appropriate segment of the start of the statement in which it appears. The location counter may be assigned to, with the restriction that the current segment may not change; furthermore, the value of "." may not decrease. If the effect of the assignment is to increase the value of "." , the required number of null bytes are generated (but see *Segments* above).

5. Statements

A source program is composed of a sequence of *statements*. Statements are separated either by new-lines or by semicolons. There are five kinds of statements: null statements, expression statements, assignment statements, string statements, and keyword statements.

Any kind of statement may be preceded by one or more labels.

5.1 Labels

There are two kinds of label: name labels and numeric labels. A name label consists of a name followed by a colon (:). The effect of a name label is to assign the current value and type of the location counter "." to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the "." value assigned changes the definition of the label.

A numeric label consists of a digit *0* to *9* followed by a colon (:). Such a label serves to define temporary symbols of the form "*n*b" and "*n*f", where *n* is the digit of the label. As in the case of name labels, a numeric label assigns the current value and type of "." to the temporary symbol. However, several numeric labels with the same digit may be used within the same assembly. References of the form "*n*f" refer to the first numeric label "*n*:" *f*orward from the reference; "*n*b" symbols refer to the first "*n* :" label *b*ackward from the reference. This sort of temporary label was introduced by Knuth [*The Art of Computer Programming, Vol I: Fundamental Algorithms*]. Such labels tend to conserve both the symbol table space of the assembler and the inventive powers of the programmer.

5.2 Null statements

A null statement is an empty statement (which may, however, have labels). A null statement is ignored by the assembler. Common examples of null statements are empty lines or lines containing only a label.

5.3 Expression statements

An expression statement consists of an arithmetic expression not beginning with a keyword. The assembler computes its (16-bit) value and places it in the output stream, together with the appropriate relocation bits.

5.4 Assignment statements

An assignment statement consists of an identifier, an equals sign (=), and an expression. The value and type of the expression are assigned to the identifier. It is not required that the type or value be the same in pass 2 as in pass 1, nor is it an error to redefine any symbol by assignment.

Any external attribute of the expression is lost across an assignment. This means that it is not possible to declare a global symbol by assigning to it, and that it is impossible to define a symbol to be offset from a non-locally defined global symbol.

As mentioned, it is permissible to assign to the location counter "`.`". It is required, however, that the type of the expression assigned be of the same type as "`.`", and it is forbidden to decrease the value of "`.`". In practice, the most common assignment to "`.`" has the form "`. = . + n`" for some number *n*; this has the effect of generating *n* null bytes.

5.5 String statements

A string statement generates a sequence of bytes containing ASCII characters. A string statement consists of a left string quote "`<`" followed by a sequence of ASCII characters not including newline, followed by a right string quote "`>`". Any of the ASCII characters may be replaced by a two-character escape sequence to represent certain non-graphic characters, as follows:

\n	NL	(012)
\s	SP	(040)
\t	HT	(011)
\e	EOT	(004)
\0	NUL	(000)
\r	CR	(015)
\a	ACK	(006)
\p	PFX	(033)
\\	\	
\>	>	

The last two are included so that the escape character and the right string quote may be represented. The same escape sequences may also be used within single- and double-character constants (see §2.3 above).

5.6 Keyword statements

Keyword statements are numerically the most common type, since most machine instructions are of this sort. A keyword statement begins with one of the many predefined keywords of the assembler; the syntax of the remainder depends on the keyword. All the keywords are listed below with the syntax they require.

6. Expressions

An expression is a sequence of symbols representing a value. Its constituents are identifiers, constants, temporary symbols, operators, and brackets. Each expression has a type.

All operators in expressions are fundamentally binary in nature; if an operand is missing on the left, a 0 of absolute type is assumed. Arithmetic is two's complement and has 16 bits of precision. All operators have equal precedence, and expressions are evaluated strictly left to right except for the effect of brackets.

6.1 Expression operators

The operators are:

(blank) when there is no operand between operands, the effect is exactly the same as if a "+" had appeared.

\+ addition

\- subtraction

* multiplication

\/ division (note that plain " / " starts a comment)

8 bitwise **and**

| bitwise **or**

\> logical right shift

\< logical left shift

% modulo

! *a* ! *b* is *a* **or** (**not** *b*); i.e., the **or** of the first operand and the one's complement of the second; most common use is as a unary.

^ result has the value of first operand and the type of the second; most often used to define new machine instructions with syntax identical to existing instructions.

Expressions may be grouped by use of square brackets " [ ] ". (Round parentheses are reserved for address modes.)

6.2 Types

The assembler deals with a number of types of expressions. Most types are attached to keywords and used to select the routine which treats that keyword. The types likely to be met explicitly are:

undefined

Upon first encounter, each symbol is undefined. It may become undefined if it is assigned an undefined expression. It is an error to attempt to assemble an undefined expression in pass 2; in pass 1, it is not (except that certain keywords require operands which are not undefined).

undefined external

A symbol which is declared **.globl** but not defined in the current assembly is an undefined external. If such a symbol is declared, the link editor *ld* must be used to load the assembler's output with another routine that defines the undefined reference.

absolute An absolute symbol is defined ultimately from a constant. Its value is unaffected by any possible future applications of the link-editor to the output file.

text The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value since the program need not be the first in the link editor's output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of " . " is text 0.

data The value of a data symbol is measured with respect to the origin of the data segment of a program. Like text symbols, the value of a data symbol may change during a subsequent link-editor run since previously loaded programs may have data segments. After the first **.data** statement, the value of " . " is data 0.

bss The value of a bss symbol is measured from the beginning of the bss segment of a program. Like text and data symbols, the value of a bss symbol may change during a subsequent link-editor run, since previously loaded programs may have bss segments. After the first **.bss** statement, the value of " . " is bss 0.

external absolute, text, data, or bss

symbols declared **.globl** but defined within an assembly as absolute, text, data, or bss symbols may be used exactly as if they were not declared **.globl**; however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

register

The symbols

r0 ... r5
fr0 ... fr5
sp
pc

are predefined as register symbols. Either they or symbols defined from them must be used to refer to the six general-purpose, six floating-point, and the 2 special-purpose machine registers. The behavior of the floating register names is identical to that of the corresponding general register names; the former are provided as a mnemonic aid.

other types

Each keyword known to the assembler has a type which is used to select the routine which processes the associated keyword statement. The behavior of such symbols when not used as keywords is the same as if they were absolute.

6.3 Type propagation in expressions

When operands are combined by expression operators, the result has a type which depends on the types of the operands and on the operator. The rules involved are complex to state but were intended to be sensible and predictable. For purposes of expression evaluation the important types are

undefined
absolute
text
data
bss
undefined external
other

The combination rules are then: If one of the operands is undefined, the result is undefined. If both operands are absolute, the result is absolute. If an absolute is combined with one of the "other types" mentioned above, or with a register expression, the result has the register or other type. As a consequence, one can refer to r3 as "r0+3". If two operands of "other type" are combined, the result has the numerically larger type An "other type" combined with an explicitly discussed type other than absolute acts like an absolute.

Further rules applying to particular operators are:

\+ If one operand is text-, data-, or bss-segment relocatable, or is an undefined external, the result has the postulated type and the other operand must be absolute.

\- If the first operand is a relocatable text-, data-, or bss-segment symbol, the second operand may be absolute (in which case the result has the type of the first operand); or the second operand may have the same type as the first (in which case the result is absolute). If the first operand is external undefined, the second must be absolute. All other combinations are illegal.

* This operator follows no other rule than that the result has the value of the first operand and the type of the second.

others

It is illegal to apply these operators to any but absolute symbols.

7. Pseudo-operations

The keywords listed below introduce statements that generate data in unusual forms or influence the later operations of the assembler. The metanotation

[stuff] ...

means that 0 or more instances of the given stuff may appear. Also, boldface tokens are literals, italic words are substitutable.

7.1 .byte *expression* [, *expression*] ...

The *expressions* in the comma-separated list are truncated to 8 bits and assembled in successive bytes. The expressions must be absolute. This statement and the string statement above are the only ones that assemble data one byte at at time.

7.2 .even

If the location counter " . " is odd, it is advanced by one so the next statement will be assembled at a word boundary.

7.3 .if *expression*

The *expression* must be absolute and defined in pass 1. If its value is nonzero, the **.if** is ignored; if zero, the statements between the **.if** and the matching **.endif** (below) are ignored. **.if** may be nested. The effect of **.if** cannot extend beyond the end of the input file in which it appears. (The statements are not totally ignored, in the following sense: .ifs and .endifs are scanned for, and moreover all names are entered in the symbol table. Thus names occurring only inside an **.if** will show up as undefined if the symbol table is listed.)

7.4 .endif

This statement marks the end of a conditionally-assembled section of code. See **.if** above.

7.5 .globl *name* [, *name*] ...

This statement makes the *names* external. If they are otherwise defined (by assignment or appearance as a label) they act within the assembly exactly as if the **.globl** statement were not given; however, the link editor *ld* may be used to combine this routine with other routines that refer these symbols.

Conversely, if the given symbols are not defined within the current assembly, the link editor can combine the output of this assembly with that of others which define the symbols. As discussed in §1, it is possible to force the assembler to make all otherwise undefined symbols external.

7.6 .text

7.7 .data

7.8 .bss

These three pseudo-operations cause the assembler to begin assembling into the text, data, or bss segment respectively. Assembly starts in the text segment. It is forbidden to assemble any code or data into the bss segment, but symbols may be defined and " . " moved about by assignment.

7.9 .comm *name* , *expression*

Provided the *name* is not defined elsewhere, this statement is equivalent to

```
.globl name
name = expression + name
```

That is, the type of *name* is "undefined external", and its value is *expression*. In fact the *name* behaves in the current assembly just like an undefined external. However, the link-editor *ld* has been special-cased so that all external symbols which are not otherwise defined, and which have a non-zero value, are defined to lie in the bss segment, and enough space is left after the symbol to hold *expression* bytes. All symbols which become defined in this way are located before all the explicitly defined bss-segment locations.

8. Machine instructions

Because of the rather complicated instruction and addressing structure of the PDP-11, the syntax of machine instruction statements is varied. Although the following sections give the syntax in detail, the machine handbooks should be consulted on the semantics.

8.1 Sources and Destinations

The syntax of general source and destination addresses is the same. Each must have one of the following forms, where *reg* is a register symbol, and *expr* is any sort of expression:

syntax	words	mode
reg	0	00+*reg*
(*reg*) +	0	20+*reg*
− (*reg*)	0	40+*reg*
expr (*reg*)	1	60+*reg*
(*reg*)	0	10+*reg*
**reg*	0	10+*reg*
* (*reg*) +	0	30+*reg*
* − (*reg*)	0	50+*reg*
* (*reg*)	1	70+*reg*
**expr* (*reg*)	1	70+*reg*
expr	1	67
$*expr*	1	27
**expr*	1	77
* $*expr*	1	37

The *words* column gives the number of address words generated; the *mode* column gives the octal address-mode number. The syntax of the address forms is identical to that in DEC assemblers, except that "*" has been substituted for "@" and "$" for "#"; the UNIX typing conventions make "@" and "#" rather inconvenient.

Notice that mode "*reg" is identical to "(reg)"; that "*(reg)" generates an index word (namely, 0); and that addresses consisting of an unadorned expression are assembled as pc-relative references independent of the type of the expression. To force a non-relative reference, the form "*$expr" can be used, but notice that further indirection is impossible.

8.3 Simple machine instructions

The following instructions are defined as absolute symbols:

clc
clv
clz
cln
sec
sev
sez
sen

They therefore require no special syntax. The PDP-11 hardware allows more than one of the "clear" class, or alternatively more than one of the "set" class to be **or**-ed together; this may be expressed as follows:

```
clc | clv
```

8.4 Branch

The following instructions take an expression as operand. The expression must lie in the same segment as the reference, cannot be undefined-external, and its value cannot differ from the current location of "." by more than 254 bytes:

br	**blos**	
bne	**bvc**	
beq	**bvs**	
bge	**bhis**	
blt	**bec**	(= **bcc**)
bgt	**bcc**	
ble	**blo**	
bpl	**bcs**	
bmi	**bes**	(= **bcs**)
bhi		

bes ("branch on error set") and **bec** ("branch on error clear") are intended to test the error bit returned by system calls (which is the c-bit).

8.5 Extended branch instructions

The following symbols are followed by an expression representing an address in the same segment as ".". If the target address is close enough, a branch-type instruction is generated; if the address is too far away, a **jmp** will be used.

jbr	**jlos**
jne	**jvc**
jeq	**jvs**
jge	**jhis**
jlt	**jec**
jgt	**jcc**
jle	**jlo**
jpl	**jcs**
jmi	**jes**
jhi	

jbr turns into a plain **jmp** if its target is too remote; the others (whose names are contructed by replacing the "b" in the branch instruction's name by "j") turn into the converse branch over a **jmp** to the target address.

8.6 Single operand instructions

The following symbols are names of single-operand machine instructions. The form of address expected is discussed in §8.1 above.

clr	sbcb
clrb	ror
com	rorb
comb	rol
inc	rolb
incb	asr
dec	asrb
decb	asl
neg	aslb
negb	jmp
adc	swab
adcb	tst
sbc	tstb

8.7 Double operand instructions

The following instructions take a general source and destination (§8.1), separated by a comma, as operands.

mov
movb
cmp
cmpb
bit
bitb
bic
bicb
bis
bisb
add
sub

8.8 Miscellaneous instructions

The following instructions have more specialized syntax. Here *reg* is a register name, *src* and *dst* a general source or destination (§8.1), and *expr* is an expression:

jsr	*reg*,*dst*	
rts	*reg*	
sys	*expr*	
ash	*src*, *reg*	(or, **als**)
ashc	*src*, *reg*	(or, **alsc**)
mul	*src*, *reg*	(or, **mpy**)
div	*src*, *reg*	(or, **dvd**)
xor	*reg*, *dst*	
sxt	*dst*	
mark	*expr*	
sob	*reg*, *expr*	

sys is another name for the **trap** instruction. It is used to code system calls. Its operand is required to be expressible in 6 bits. The expression in **mark** must be expressible in six bits, and the expression in **sob** must be in the same segment as ".", must not be external-undefined, must be less than ".", and must be within 510 bytes of ".".

8.9 Floating-point unit instructions

The following floating-point operations are defined, with syntax as indicated:

cfcc		
setf		
setd		
seti		
setl		
clrf	*fdst*	
negf	*fdst*	
absf	*fdst*	
tstf	*fsrc*	
movf	*fsrc, freg*	(= ldf)
movf	*freg, fdst*	(= stf)
movif	*src, freg*	(= ldcif)
movfi	*freg, dst*	(= stcfi)
movof	*fsrc, freg*	(= ldcdf)
movfo	*freg, fdst*	(= stcfd)
movie	*src, freg*	(= ldexp)
movei	*freg, dst*	(= stexp)
addf	*fsrc, freg*	
subf	*fsrc, freg*	
mulf	*fsrc, freg*	
divf	*fsrc, freg*	
cmpf	*fsrc, freg*	
modf	*fsrc, freg*	
ldfps	*src*	
stfps	*dst*	
stst	*dst*	

fsrc, *fdst*, and *freg* mean floating-point source, destination, and register respectively. Their syntax is identical to that for their non-floating counterparts, but note that only floating registers 0-3 can be a *freg*.

The names of several of the operations have been changed to bring out an analogy with certain fixed-point instructions. The only strange case is **movf**, which turns into either **stf** or **ldf** depending respectively on whether its first operand is or is not a register. Warning: **ldf** sets the floating condition codes, **stf** does not.

9. Other symbols

9.1 ..

The symbol "." is the *relocation counter*. Just before each assembled word is placed in the output stream, the current value of this symbol is added to the word if the word refers to a text, data or bss segment location. If the output word is a pc-relative address word that refers to an absolute location, the value of "." is subtracted.

Thus the value of "." can be taken to mean the starting memory location of the program. The initial value of "." is 0.

The value of "." may be changed by assignment. Such a course of action is sometimes necessary, but the consequences should be carefully thought out. It is particularly ticklish to change "." midway in an assembly or to do so in a program which will be treated by the loader, which has its own notions of ".".

9.2 System calls

System call names are not predefined. They may be found in the file */usr/include/sys.s*

10. Diagnostics

When an input file cannot be read, its name followed by a question mark is typed and assembly ceases. When syntactic or semantic errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

)	parentheses error
]	parentheses error
>	string not terminated properly
*	indirection (*) used illegally
.	illegal assignment to " . "
A	error in address
B	branch address is odd or too remote
E	error in expression
F	error in local ("f" or "b") type symbol
G	garbage (unknown) character
I	end of file inside an .if
M	multiply defined symbol as label
O	word quantity assembled at odd address
P	phase error— " . " different in pass 1 and 2
R	relocation error
U	undefined symbol
X	syntax error

Reprinted from
Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction
SIGPLAN Notices 17, 6 (June 1982), pp 120-126

gprof: a Call Graph Execution Profiler[1]

by
Susan L. Graham
Peter B. Kessler
Marshall K. McKusick

Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, California 94720

Abstract

Large complex programs are composed of many small routines that implement abstractions for the routines that call them. To be useful, an execution profiler must attribute execution time in a way that is significant for the logical structure of a program as well as for its textual decomposition. This data must then be displayed to the user in a convenient and informative way. The **gprof** profiler accounts for the running time of called routines in the running time of the routines that call them. The design and use of this profiler is described.

1. Programs to be Profiled

Software research environments normally include many large programs both for production use and for experimental investigation. These programs are typically modular, in accordance with generally accepted principles of good program design. Often they consist of numerous small routines that implement various abstractions. Sometimes such large programs are written by one programmer who has understood the requirements for these abstractions, and has programmed them appropriately. More frequently the program has had multiple authors and has evolved over time, changing the demands placed on the implementation of the abstractions without changing the implementation itself. Finally, the program may be assembled from a library of abstraction implementations unexamined by the programmer.

Once a large program is executable, it is often desirable to increase its speed, especially if small portions of the program are found to dominate its execution time. The purpose of the **gprof** profiling tool is to help the user evaluate alternative implementations of abstractions. We developed this tool in response to our efforts to improve a code generator we were writing [Graham82].

The **gprof** design takes advantage of the fact that the programs to be measured are large, structured and hierarchical. We provide a profile in which the execution time for a set of routines that implement an abstraction is collected and charged to that abstraction. The profile can be used to compare and assess the costs of various implementations.

The profiler can be linked into a program without special planning by the programmer. The overhead for using gprof is low; both in terms of added execution time and in the volume of profiling information recorded.

2. Types of Profiling

There are several different uses for program profiles, and each may require different information from the profiles, or different presentation of the information. We distinguish two broad categories of profiles: those that present counts of statement or routine invocations, and those that display timing information about statements or routines. Counts are typically presented in tabular form, often in parallel with a listing of the source code. Timing information could be similarly presented; but more than one measure of time might be associated with each statement or routine. For example, in the framework used by **gprof** each profiled segment would display two times: one for the time used by the segment itself, and another for the time inherited from code segments it invokes.

Execution counts are used in many different contexts. The exact number of times a routine or statement is activated can be used to determine if an algorithm is performing as expected. Cursory inspection of such counters may show algorithms whose complexity is unsuited to the task at hand. Careful interpretation of counters can often suggest improvements to acceptable algorithms. Precise examination can uncover subtle errors in an

[1]This work was supported by grant MCS80-05144 from the National Science Foundation.

algorithm. At this level, profiling counters are similar to debugging statements whose purpose is to show the number of times a piece of code is executed. Another view of such counters is as boolean values. One may be interested that a portion of code has executed at all, for exhaustive testing, or to check that one implementation of an abstraction completely replaces a previous one.

Execution counts are not necessarily proportional to the amount of time required to execute the routine or statement. Further, the execution time of a routine will not be the same for all calls on the routine. The criteria for establishing execution time must be decided. If a routine implements an abstraction by invoking other abstractions, the time spent in the routine will not accurately reflect the time required by the abstraction it implements. Similarly, if an abstraction is implemented by several routines the time required by the abstraction will be distributed across those routines.

Given the execution time of individual routines, **gprof** accounts to each routine the time spent for it by the routines it invokes. This accounting is done by assembling a *call graph* with nodes that are the routines of the program and directed arcs that represent calls from call sites to routines. We distinguish among three different call graphs for a program. The *complete call graph* incorporates all routines and all potential arcs, including arcs that represent calls to functional parameters or functional variables. This graph contains the other two graphs as subgraphs. The *static call graph* includes all routines and all possible arcs that are not calls to functional parameters or variables. The *dynamic call graph* includes only those routines and arcs traversed by the profiled execution of the program. This graph need not include all routines, nor need it include all potential arcs between the routines it covers. It may, however, include arcs to functional parameters or variables that the static call graph may omit. The static call graph can be determined from the (static) program text. The dynamic call graph is determined only by profiling an execution of the program. The complete call graph for a monolithic program could be determined by data flow analysis techniques. The complete call graph for programs that change during execution, by modifying themselves or dynamically loading or overlaying code, may never be determinable. Both the static call graph and the dynamic call graph are used by **gprof**, but it does not search for the complete call graph.

3. Gathering Profile Data

Routine calls or statement executions can be measured by having a compiler augment the code at strategic points. The additions can be inline increments to counters [Knuth71] [Satterthwaite72] [Joy79] or calls to monitoring routines [Unix]. The counter increment overhead is low, and is suitable for profiling statements. A call of the monitoring routine has an overhead comparable with a call of a regular routine, and is therefore only suited to profiling on a routine by routine basis. However, the monitoring routine solution has certain advantages. Whatever counters are needed by the monitoring routine can be managed by the monitoring routine itself, rather than being distributed around the code. In particular, a monitoring routine can easily be called from separately compiled programs. In addition, different monitoring routines can be linked into the program being measured to assemble different profiling data without having to change the compiler or recompile the program. We have exploited this approach: our compilers for C, Fortran77, and Pascal can insert calls to a monitoring routine in the prologue for each routine. Use of the monitoring routine requires no planning on part of a programmer other than to request that augmented routine prologues be produced during compilation.

We are interested in gathering three pieces of information during program execution: call counts and execution times for each profiled routine, and the arcs of the dynamic call graph traversed by this execution of the program. By post-processing of this data we can build the dynamic call graph for this execution of the program and propagate times along the edges of this graph to attribute times for routines to the routines that invoke them.

Gathering of the profiling information should not greatly interfere with the running of the program. Thus, the monitoring routine must not produce trace output each time it is invoked. The volume of data thus produced would be unmanageably large, and the time required to record it would overwhelm the running time of most programs. Similarly, the monitoring routine can not do the analysis of the profiling data (e.g. assembling the call graph, propagating times around it, discovering cycles, etc.) during program execution. Our solution is to gather profiling data in memory during program execution and to condense it to a file as the profiled program exits. This file is then processed by a separate program to produce the listing of the profile data. An advantage of this approach is that the profile data for several executions of a program can be combined by the post-processing to provide a profile of many executions.

The execution time monitoring consists of three parts. The first part allocates and initializes the runtime monitoring data structures before the program begins execution. The second part is the monitoring routine invoked from the prologue of each profiled routine. The third part condenses the data structures and writes them to a file as the program terminates. The monitoring routine is discussed in detail in the following sections.

3.1. Execution Counts

The **gprof** monitoring routine counts the number of times each profiled routine is called. The monitoring routine also records the arc in the call graph that activated the profiled routine. The count is associated with the arc in the call graph rather than with the routine. Call counts for routines can then be determined by summing the counts on arcs directed into that routine. In a machine-dependent

fashion, the monitoring routine notes its own return address. This address is in the prologue of some profiled routine that is the destination of an arc in the dynamic call graph. The monitoring routine also discovers the return address for that routine, thus identifying the call site, or source of the arc. The source of the arc is in the *caller*, and the destination is in the *callee*. For example, if a routine A calls a routine B, A is the caller, and B is the callee. The prologue of B will include a call to the monitoring routine that will note the arc from A to B and either initialize or increment a counter for that arc.

One can not afford to have the monitoring routine output tracing information as each arc is identified. Therefore, the monitoring routine maintains a table of all the arcs discovered, with counts of the numbers of times each is traversed during execution. This table is accessed once per routine call. Access to it must be as fast as possible so as not to overwhelm the time required to execute the program.

Our solution is to access the table through a hash table. We use the call site as the primary key with the callee address being the secondary key. Since each call site typically calls only one callee, we can reduce (usually to one) the number of minor lookups based on the callee. Another alternative would use the callee as the primary key and the call site as the secondary key. Such an organization has the advantage of associating callers with callees, at the expense of longer lookups in the monitoring routine. We are fortunate to be running in a virtual memory environment, and (for the sake of speed) were able to allocate enough space for the primary hash table to allow a one-to-one mapping from call site addresses to the primary hash table. Thus our hash function is trivial to calculate and collisions occur only for call sites that call multiple destinations (e.g. functional parameters and functional variables). A one level hash function using both call site and callee would result in an unreasonably large hash table. Further, the number of dynamic call sites and callees is not known during execution of the profiled program.

Not all callers and callees can be identified by the monitoring routine. Routines that were compiled without the profiling augmentations will not call the monitoring routine as part of their prologue, and thus no arcs will be recorded whose destinations are in these routines. One need not profile all the routines in a program. Routines that are not profiled run at full speed. Certain routines, notably exception handlers, are invoked by non-standard calling sequences. Thus the monitoring routine may know the destination of an arc (the callee), but find it difficult or impossible to determine the source of the arc (the caller). Often in these cases the apparent source of the arc is not a call site at all. Such anomalous invocations are declared "spontaneous".

3.2. Execution Times

The execution times for routines can be gathered in at least two ways. One method measures the execution time of a routine by measuring the elapsed time from routine entry to routine exit. Unfortunately, time measurement is complicated on time-sharing systems by the time-slicing of the program. A second method samples the value of the program counter at some interval, and infers execution time from the distribution of the samples within the program. This technique is particularly suited to time-sharing systems, where the time-slicing can serve as the basis for sampling the program counter. Notice that, whereas the first method could provide exact timings, the second is inherently a statistical approximation.

The sampling method need not require support from the operating system: all that is needed is the ability to set and respond to "alarm clock" interrupts that run relative to program time. It is imperative that the intervals be uniform since the sampling of the program counter rather than the duration of the interval is the basis of the distribution. If sampling is done too often, the interruptions to sample the program counter will overwhelm the running of the profiled program. On the other hand, the program must run for enough sampled intervals that the distribution of the samples accurately represents the distribution of time for the execution of the program. As with routine call tracing, the monitoring routine can not afford to output information for each program counter sample. In our computing environment, the operating system can provide a histogram of the location of the program counter at the end of each clock tick (1/60th of a second) in which a program runs. The histogram is assembled in memory as the program runs. This facility is enabled by our monitoring routine. We have adjusted the granularity of the histogram so that program counter values map one-to-one onto the histogram. We make the simplifying assumption that all calls to a specific routine require the same amount of time to execute. This assumption may disguise that some calls (or worse, some call sites) always invoke a routine such that its execution is faster (or slower) than the average time for that routine.

When the profiled program terminates, the arc table and the histogram of program counter samples are written to a file. The arc table is condensed to consist of the source and destination addresses of the arc and the count of the number of times the arc was traversed by this execution of the program. The recorded histogram consists of counters of the number of times the program counter was found to be in each of the ranges covered by the histogram. The ranges themselves are summarized as a lower and upper bound and a step size.

4. Post Processing

Having gathered the arcs of the call graph and timing information for an execution of the program, we are interested in attributing the time for each routine to the routines that call it. We build a dynamic call graph with arcs from caller to callee, and propagate time from descendants to ancestors by topologically sorting the call graph. Time

propagation is performed from the leaves of the call graph toward the roots, according to the order assigned by a topological numbering algorithm. The topological numbering ensures that all edges in the graph go from higher numbered nodes to lower numbered nodes. An example is given in Figure 1. If we propagate time from nodes in the order assigned by the algorithm, execution time can be propagated from descendants to ancestors after a single traversal of each arc in the call graph. Each parent receives some fraction of a child's time. Thus time is charged to the caller in addition to being charged to the callee.

Let C_e be the number of calls to some routine, e, and C_e^r be the number of calls from a caller r to a callee e. Since we are assuming each call to a routine takes the average amount of time for all calls to that routine, the caller is accountable for C_e^r / C_e of the time spent by the callee. Let the S_e be the *selftime* of a routine, e. The selftime of a routine can be determined from the timing information gathered during profiled program execution. The total time, T_r, we wish to account to a routine r, is then given by the recurrence equation:

$$T_r = S_r + \sum_{r \text{ CALLS } e} T_e \times \frac{C_e^r}{C_e}$$

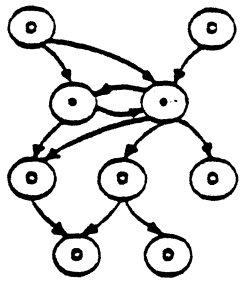
where r CALLS e is a relation showing all routines e called by a routine r. This relation is easily available from the call graph.

However, if the execution contains recursive calls, the call graph has cycles that cannot be topologically sorted. In these cases, we discover strongly-connected components in the call graph, treat each such component as a single node, and then sort the resulting graph. We use a variation of Tarjan's strongly-connected components algorithm that discovers strongly-connected components as it is assigning topological order numbers [Tarjan72].

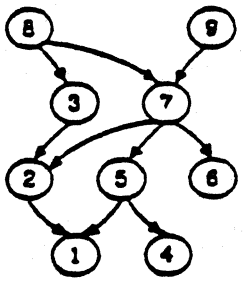
Time propagation within strongly connected components is a problem. For example, a self-recursive routine (a trivial cycle in the call graph) is accountable for all the time it uses in all its recursive instantiations. In our scheme, this time should be shared among its call graph parents. The arcs from a routine to itself are of interest, but do not participate in time propagation. Thus the simple equation for time propagation does not work within strongly connected components. Time is not propagated from one member of a cycle to another, since, by definition, this involves propagating time from a routine to itself. In addition, children of one member of a cycle must be considered children of all members of the cycle. Similarly, parents of one member of the cycle must inherit all members of the cycle as descendants. It is for these reasons that we collapse connected components. Our solution collects all members of a cycle together, summing the time and call counts for all members. All calls into the cycle are made to share the total time of the cycle, and all descendants of the cycle propagate time into the cycle as a whole. Calls among the members of the cycle do not propagate any time, though they are listed in the call graph profile.

Figure 2 shows a modified version of the call graph of Figure 1, in which the nodes labelled 3 and 7 in Figure 1 are mutually recursive. The topologically sorted graph after the cycle is collapsed is given in Figure 3.

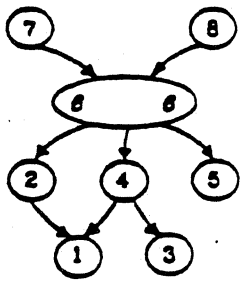
Since the technique described above only collects the dynamic call graph, and the program typically does not call every routine on each execution, different executions can introduce different cycles in the dynamic call graph. Since cycles often have a significant effect on time propagation, it is desirable to incorporate the static call graph so that cycles will have the same members regardless of how the program runs.



Cycle to be collapsed.
Figure 2.



Topological ordering
Figure 1.



Topological numbering after cycle collapsing.
Figure 3.

The static call graph can be constructed from the source text of the program. However, discovering the static call graph from the source text would require two moderately difficult steps: finding the source text for the program (which may not be available), and scanning and parsing that text, which may be in any one of several languages.

In our programming system, the static calling information is also contained in the executable version of the program, which we already have available, and which is in language-independent form. One can examine the instructions in the object program, looking for calls to routines, and note which routines can be called. This technique allows us to add arcs to those already in the dynamic call graph. If a statically discovered arc already exists in the dynamic call graph, no action is required. Statically discovered arcs that do not exist in the dynamic call graph are added to the graph with a traversal count of zero. Thus they are never responsible for any time propagation. However, they may affect the structure of the graph. Since they may complete strongly connected components, the static call graph construction is done before topological ordering.

5. Data Presentation

The data is presented to the user in two different formats. The first presentation simply lists the routines without regard to the amount of time their descendants use. The second presentation incorporates the call graph of the program.

5.1. The Flat Profile

The flat profile consists of a list of all the routines that are called during execution of the program, with the count of the number of times they are called and the number of seconds of execution time for which they are themselves accountable. The routines are listed in decreasing order of execution time. A list of the routines that are never called during execution of the program is also available to verify that nothing important is omitted by this execution. The flat profile gives a quick overview of the routines that are used, and shows the routines that are themselves responsible for large fractions of the execution time. In practice, this profile usually shows that no single function is overwhelmingly responsible for the total time of the program. Notice that for this profile, the individual times sum to the total execution time.

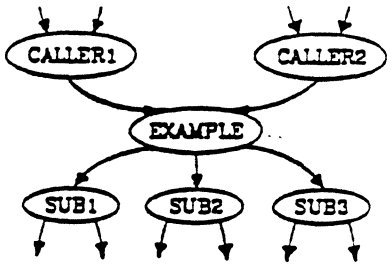
5.2. The Call Graph Profile

Ideally, we would like to print the call graph of the program, but we are limited by the two-dimensional nature of our output devices. We cannot assume that a call graph is planar, and even if it is, that we can print a planar version of it. Instead, we choose to list each routine, together with information about the routines that are its direct parents and children. This listing presents a window into the call graph. Based on our experience, both parent information and child information is important, and should be available without searching through the output.

The major entries of the call graph profile are the entries from the flat profile, augmented by the time propagated to each routine from its descendants. This profile is sorted by the sum of the time for the routine itself plus the time inherited from its descendants. The profile shows which of the higher level routines spend large portions of the total execution time in the routines that they call. For each routine, we show the amount of time passed by each child to the routine, which includes time for the child itself and for the descendants of the child (and thus the descendants of the routine). We also show the percentage these times represent of the total time accounted to the child. Similarly, the parents of each routine are listed, along with time, and percentage of total routine time, propagated to each one.

Cycles are handled as single entities. The cycle as a whole is shown as though it were a single routine, except that members of the cycle are listed in place of the children. Although the number of calls of each member from within the cycle are shown, they do not affect time propagation. When a child is a member of a cycle, the time shown is the appropriate fraction of the time for the whole cycle. Self-recursive routines have their calls broken down into calls from the outside and self-recursive calls. Only the outside calls affect the propagation of time.

The following example is a typical fragment of a call graph.



The entry in the call graph profile listing for this example is shown in Figure 4.

The entry is for routine EXAMPLE, which has the Caller routines as its parents, and the Sub routines as its children. The reader should keep in mind that all information is given *with respect to EXAMPLE*. The index in the first column shows that EXAMPLE is the second entry in the profile listing. The EXAMPLE routine is called ten times, four times by CALLER1, and six times by CALLER2. Consequently 40% of EXAMPLE's time is propagated to CALLER1, and 60% of EXAMPLE's time is propagated to CALLER2. The self and descendant fields of the parents show the amount of self and descendant time EXAMPLE propagates to them (but not the time used by the parents directly). Note that EXAMPLE calls itself recursively four times. The routine EXAMPLE calls routine SUB1 twenty times, SUB2 once, and never calls SUB3. Since SUB2 is called a total of five times, 20% of its self and descendant time is propagated to EXAMPLE's descendant time field. Because SUB1 is a

index	%time	self	descendants	called/total called+self called/total	parents name children	index
		0.20	1.20	4/10	CALLER1	[7]
		0.30	1.80	6/10	CALLER2	[1]
[2]	41.5	0.50	3.00	10+4	EXAMPLE	[2]
		1.50	1.00	20/40	SUB1 <cycle1>	[4]
		0.00	0.50	1/5	SUB2	[9]
		0.00	0.00	0/5	SUB3	[11]

Profile entry for EXAMPLE.
Figure 4.

member of *cycle 1*, the self and descendant times and call count fraction are those for the cycle as a whole. Since cycle 1 is called a total of forty times (not counting calls among members of the cycle), it propagates 50% of the cycle's self and descendant time to EXAMPLE's descendant time field. Finally each name is followed by an index that shows where on the listing to find the entry for that routine.

6. Using the Profiles

The profiler is a useful tool for improving a set of routines that implement an abstraction. It can be helpful in identifying poorly coded routines, and in evaluating the new algorithms and code that replace them. Taking full advantage of the profiler requires a careful examination of the call graph profile, and a thorough knowledge of the abstractions underlying the program.

The easiest optimization that can be performed is a small change to a control construct or data structure that improves the running time of the program. An obvious starting point is a routine that is called many times. For example, suppose an output routine is the only parent of a routine that formats the data. If this format routine is expanded inline in the output routine, the overhead of a function call and return can be saved for each datum that needs to be formatted.

The drawback to inline expansion is that the data abstractions in the program may become less parameterized, hence less clearly defined. The profiling will also become less useful since the loss of routines will make its output more granular. For example, if the symbol table functions "lookup", "insert", and "delete" are all merged into a single parameterized routine, it will be impossible to determine the costs of any one of these individual functions from the profile.

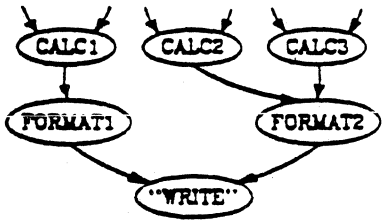
Further potential for optimization lies in routines that implement data abstractions whose total execution time is long. For example, a lookup routine might be called only a few times, but use an inefficient linear search algorithm, that might be replaced with a binary search. Alternately, the discovery that a rehashing function is being called excessively, can lead to a different hash function or a larger hash table. If the data abstraction function cannot easily be speeded up, it may be advantageous to cache its results, and eliminate the need to rerun it for identical inputs. These and other ideas for program improvement are discussed in [Bentley81].

This tool is best used in an iterative approach: profiling the program, eliminating one bottleneck, then finding some other part of the program that begins to dominate execution time. For instance, we have used gprof on itself; eliminating, rewriting, and inline expanding routines, until reading data files (hardly a target for optimization!) represents the dominating factor in its execution time.

Certain types of programs are not easily analyzed by gprof. They are typified by programs that exhibit a large degree of recursion, such as recursive descent compilers. The problem is that most of the major routines are grouped into a single monolithic cycle. As in the symbol table abstraction that is placed in one routine, it is impossible to distinguish which members of the cycle are responsible for the execution time. Unfortunately there are no easy modifications to these programs that make them amenable to analysis.

A completely different use of the profiler is to analyze the control flow of an unfamiliar program. If you receive a program from another user that you need to modify in some small way, it is often unclear where the changes need to be made. By running the program on an example and then using gprof, you can get a view of the structure of the program.

Consider an example in which you need to change the output format of the program. For purposes of this example suppose that the call graph of the output portion of the program has the following structure:



Initially you look through the gprof output for the system call "WRITE". The format routine you will need to change is probably among the parents of the "WRITE" procedure. The next step is to look at the profile entry for each of parents of "WRITE", in this example either "FORMAT1" or "FORMAT2", to determine which one to change. Each format routine will have one or more parents, in this example "CALC1", "CALC2", and "CALC3". By inspecting the source code for each of these routines you can

determine which format routine generates the output that you wish to modify. Since the gprof entry shows all the potential calls to the format routine you intend to change, you can determine if your modifications will affect output that should be left alone. If you desire to change the output of "CALC2", but not "CALC3", then formatting routine "FORMAT2" needs to be split into two separate routines, one of which implements the new format. You can then retarget just the call by "CALC2" that needs the new format. It should be noted that the static call information is particularly useful here since the test case you run probably will not exercise the entire program.

7. Conclusions

We have created a profiler that aids in the evaluation of modular programs. For each routine in the program, the profile shows the extent to which that routine helps support various abstractions, and how that routine uses other abstractions. The profile accurately assesses the cost of routines at all levels of the program decomposition. The profiler is easily used, and can be compiled into the program without any prior planning by the programmer. It adds only five to thirty percent execution overhead to the program being profiled, produces no additional output until after the program finishes, and allows the program to be measured in its actual environment. Finally, the profiler runs on a time-sharing system using only the normal services provided by the operating system and compilers.

8. References

[Bentley81]
Bentley, J. L., "Writing Efficient Code", Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, CMU-CS-81-116, 1981.

[Graham82]
Graham, S. L., Henry, R. R., Schulman, R. A., "An Experiment in Table Driven Code Generation", SIGPLAN '82 Symposium on Compiler Construction, June, 1982.

[Joy79]
Joy, W. N., Graham, S. L., Haley, C. B. "Berkeley Pascal User's Manual", Version 1.1, Computer Science Division University of California, Berkeley, CA. April 1979.

[Knuth71]
Knuth, D. E. "An empirical study of FORTRAN programs", Software - Practice and Experience, 1, 105-133. 1971

[Satterthwaite72]
Satterthwaite, E. "Debugging Tools for High Level Languages", Software - Practice and Experience, 2, 197-217, 1972

[Tarjan72]
Tarjan, R. E., "Depth first search and linear graph algorithm," *SIAM J. Computing* 1:2, 146-160, 1972.

[Unix]
Unix Programmer's Manual, "**prof** command", section 1, Bell Laboratories, Murray Hill, NJ. January 1979.

Hints on Configuring VAX* Systems for UNIX†

Revised for 4.2BSD: March 15, 1983

Bob Kridle

Computer Systems Support Group
U. C. Berkeley
kridle@berkeley, ucbvax!kridle

Sam Leffler

Computer Systems Research Group
U. C. Berkeley
sam@berkeley, ucbvax!sam

ABSTRACT

This document reflects our experiences and opinions in configuring over thirty VAXes to run UNIX† over the last five years.

Our prime considerations in choosing equipment are:

- Cost
- Performance
- Reliability
- Maintainability and maintenance cost
- Delivery time
- Redundancy of the system
- Conservation of space, power, and cooling resources

We consider components individually and then describe several system packages built from these components, emphasizing independently single-source systems, minimization of cost, and maximal expansion capability.

† UNIX is a trademark of Bell Laboratories.
* VAX, VMS, MASSBUS, and UNIBUS are trademarks of Digital Equipment Corporation.

DISCLAIMER

This documents reflects our *personal opinions.* We are responsible for software and hardware support of VAX systems, and the recommendations we give reflect what we would do. We are careful to note the equipment that we recommend but are not using; we recommend no second-vendor equipment that is not known to be in use successfully at several UNIX sites. In any case you may get a lemon, no matter what you buy. All we promise is that this is what we believe. Let us know what you find out.

We have little familiarity with VMS. Recommendations made here should not be construed to be applicable to any operating system other than UNIX. We have often adapted UNIX to these devices in a way that may not be possible with other operating systems.

Finally, note that we have not written this document solely to share the knowledge we have acquired with you; we have written it because we do not have the time to talk to everyone who needs this information. Please do not call us to confirm the information here or to ask questions about our opinions. We would like to hear of your experiences, or learn of mistakes in this document or products that we know nothing about, but do not have time to chat about the information that is given here. We do welcome electronic mail sent to our addresses as shown on the first page.

PREFACE TO 1983 REVISION

The VAX/UNIX Community has grown considerably since the last revision of this paper in mid-1981. It is safe to say that 4BSD systems represent a substantial portion of all the VAXes sold. Both the hardware available for building good VAX/UNIX systems and the support services available to those doing the building have increased. A significant portion of the systems now being built are being used in business and private research environments in addition to those in the traditional academic UNIX strongholds. Many of these systems are based on binary licenses rather than the educational source license so familiar at universities.

We hope this document will be of use to a variety of potential VAX/UNIX users. Even the purchaser who chooses to buy a complete system from DEC or a mixed vendor system integrator should be assisted in making a better choice by an increased knowledge of the integration decisions. In the area of hardware, you will find that the sections on disks, network interfaces, and printers have been heavily reworked. In addition to the traditional emphasis on hardware selection, in this revision we try to at least provide pointers to some of the other relevant services available. A list of hardware integrators we are aware of who have developed expertise in the UNIX area is included. In addition, some suppliers of 4BSD software are included.

This revision takes a different policy regarding the inclusion of specific prices for hardware: *No explict prices are included.* We do ocassionally include derived costs such as the cost per Megabyte for a particular size disk system. The are only meant to be approximate and are calculated from current quantity one list prices.

It has been our experience that the inclusion of prices made the release of this paper much more difficult without significantly increasing its value. We were obligated to pass the paper by every hardware vendor for price checking and even so the contents remained accurate for only a brief period. In addition, regional pricing and discounting policys vary widely. We have decided that rather than depend on us for prices, the reader should narrow their range of choices as much as possible and then do some hard bargaining with their suppliers. "Every-

body gets a discount."

OVERVIEW

We first discuss components, listing the alternatives we have tried and sometimes a few we have not, and then discuss system packages. We buy a substantial portion of our equipment from vendors other than DEC. The reasons for choosing second vendor equipment are usually some combination of more current technology, lower cost for equivalent equipment or shorter delivery time.

We do not consider devices that have proven unreliable or whose performance we consider inadequate.* In addition, there are many devices that we have no experience with. As a general rule, every new peripheral has required a non-trivial amount of leg work to get up to speed. We suggest using only peripherals that have been previously used successfully on *the type of VAX you are configuring* (780, 750 or 730) or demanding a substantial (50-100%) discount for being a guinea pig. Be especially careful of UNIBUS† interfaces. Almost every manufacturer of a UNIBUS widget now includes the VAX as a machine on which his device will work. Some of these devices have still not been well tested in this situation. These often will not work without substantial modification.

System buyers without ready access to an in-house hardware staff should consider carefully the option of buying as much DEC equipment as possible. If you have the money and time required to do this, there are some strong advantages. Our DEC equipment has, in general, proven somewhat more reliable than the equivalent alternate vendor equipment. Time from equipment delivery to running system is also usually shorter. DEC field service in our area is excellent. Outside service available for non-DEC peripherals is spotty at best.

For smaller installations this option should be carefully considered. It is easier if you can call one party for all your problems, if you can afford it. At Berkeley, we are well past the inventory level where self maintenance begins to pay off even on all DEC systems, so this is not a consideration. One of us (Kridle) manages our local hardware support group.

Unfortunately, the limited selection of configurations currently available sometimes make the all-DEC choice difficult. This is especially true of the smaller configurations as DEC's bottom end peripherals are less satisfactory for UNIX. We say this not just for monetary reasons; functionally and aesthetically we would prefer to have neither the RK07 disk nor the TS11 tape unit in any system we have to deal with.

We recommend getting field service at least on your CPU for the first year. It has paid off for us in the cost of parts alone. You can drop the contract after the engineering changes have tapered off and most of the infant failures have occurred. DEC requires a certain amount of its peripheral equipment on the machine to qualify for field service. We understand that it is company policy not to provide a maintenance contract for a system without a DEC mass storage peripheral. If you intend to purchase a maintenance contract, be certain that your local field

---

* An exception to this rule is made where we have yet to find any satisfactory devices in a particular category. In these instances we have indicated our reservations about the existing choices in the hopes that new products will address the problems we believe are important. The reader should realize that if a vendor's equipment has been mentioned in a negative light it indicates we at least thought highly enough of it to evaluate it seriously. We are not trying to damage any company's reputation, merely insure that important information is shared equally.

service is willing to support at least the DEC equipment you buy.

BANDWIDTH CONSIDERATIONS

Evaluation of the data transfer capacities between the various parts of VAX systems is a complex task that plays a critical part in system configuration. Unfortunately, there is a tremendous amount of misinformation available on this subject and little useful hard data. We have made many measurements and are always in the process of making more. What we currently know follows.

The 11/780 UNIBUS adapter is the device most frequently shrouded in confusion. DEC documents variously give the bandwidth at between 1.2 MB/sec and 1.5 MB/sec when transferring through a buffered data path. We are not aware of any specifications for the unbuffered data path but have not been able to use it with some devices as slow as 50 KB/sec. One experiment we conducted involved examining the UNIBUS protocol lines with a scope while constantly transferring from a disk drive. We observed that while the drive was transferring at an average rate of about 1.2 MB/sec the UNIBUS was close to one hundred precent busy. This test was conducted on an otherwise idle system. No other devices were active on the UNIBUS and large disk transfers (cylinders) reduced any register set up time to a minimum. We conclude from this that 1.2 MB/sec is the *absolute maximum* transfer rate possible through a 11/780 UNIBUS adapter. Our observations showed that the largest delays while transferring data occurred while the buffered data path was being loaded or unloaded from the SBI. Since the UBA is controlled by a micro sequencer that is also involved in other UBA activities such as processing interrupts, we suspect that on an active UBA this bandwidth may be somewhat reduced.

Measurements of the available throughput to and from the 4.2BSD file system indicate a significant difference between disks running on the native processor bus (CMI or SBI) and those running on the UNIBUS. Average data rates are consistently lower on disks residing on the UNIBUS, even when the controller provides a few sectors of buffering. This leads us to believe that when average reads are 4-8 Kilobytes (the average block size of a 4.2BSD file system), most UNIBUS controllers will fall behind and eventually lose a revolution. This does not, however, seem to occur with the UDA50 UNIBUS controller as it has a much larger amount (16 Kbytes) of buffering†.

There are troublesome devices that cannot buffer enough data to guarantee that the maximum size record can always be transferred (6250bpi tape drives), or do not buffer an adequate amount of data (RK07 disk controller). To handle these devices UNIX provides a software interlock mechanism that prevents excess UBA contention.

The MASSBUS adapters are specified to have a higher potential bandwidth of 2.5 MB/sec. Since they are selector channels that allow only one device to transfer data at a time, the realized bandwidth is limited to the rate of the fastest device. The fastest devices currently available from DEC for 11/750 systems or 11/780 systems with a single memory controller transfer at 1.3 MB/sec. Large 11/780 systems with two memory controllers and interleaved memory may run RP07 disk drives that then transfer data at 2.2 MB/sec.* An interesting bandwidth limit may be established by the memory controller particularly on 11/780s. We suspect that the CPU may be slowed considerably by memory contention when two disk channels are being used simultaneously. This should be alleviated by using interleaved memory controllers.

The appendix to the *VAX Hardware Handbook* titled "System Throughput Considerations" seems to bear out these impressions and should be read carefully by anyone hoping to understand the consequences for VAX applications involving high bandwidth input or output. If we had data intensive applications we would seriously consider the use of RP07 disks (and

† A few of the initial UDA50 controllers were delivered with only 4 Kbytes of buffering. Avoid these.

* On machines with only one memory controller the RP07 hardware is arranged to transfer at 1.3 MB/sec.

interleaved memory controllers) because of the resultant higher burst transfer rate; this will be discussed further below.

MEMORY

All VAXes are sold with at least the minimum amount of DEC memory adequate to run diagnostics. Additional memory is the lowest risk alternate vendor choice. We buy the remainder of our 780 memory from Mostek, National Semiconductor or Trendata.* This area is extremely price competitive and there are at least six possible vendors. By all means, ask for competitive quotes. Assure yourself, however, that you are not the first customer for a new vendor.

Add in memory for the VAX 750 is a newer item and prices are not as low. However, this memory is almost identical to the 11/70 MK11 memory and several vendors have managed to build this product by modifying their previous 11/70 add-in product. Trendata also has 1 Mbyte 64K RAM modules for both 750s and 730s.

Small quantities (one to two megabytes) are usually available off the shelf. Large quantities (4 megabytes and up) have taken less than 30 days.

For the 11/780 memory, the RAM chips are socketed, and two replacement chips per board are supplied by all vendors we mention; You can pull out the board and replace the chip at your leisure. Since single bit errors are corrected this has never involved any unexpected down time for us. There is at least a one year return to factory agreement on the boards, included in the purchase price. Out of warranty repairs are said to typically cost less than $300. We have returned only one board to the plant in about 30 board years.

When purchased from DEC, memory is much more expensive for any of the machines. Maintenance on a 1 Megabyte DEC memory module is $179 per month with board replacement through field service. The boards are not socketed. Delivery times on memory from DEC have typically been substantially longer than times from second vendors.

If you are going to have more than 4 megabytes of memory on your 780 you will need a CPU expansion cabinet and a second memory controller that includes a second half-megabyte of DEC memory.

There are two models of 11/750 memory controllers and backplanes around. The one currently being manufactured by DEC can be filled with either quarter Megabyte or full Megabyte modules for a maximum capacity of 8 Megabytes‡. The older memory controller and backplane can be populated with only quarter Megabyte modules for a total capacity of 2 Megabytes. To make matters even more complicated, 750s exist which have the newer style backplane and the older controller. These too will only hold 2 Megabytes of memory. The smaller capacity system can be upgraded to the larger one, but this is quite expensive; check with DEC before buying one, or be sure that you will be satisfied with a maximum of 2 Megabytes.

DISKS†

The area of disks and disk controllers is one which has seen a great deal of change since the last revision of this paper in mid 1981. At that time we had no experience with Winchester technology disk drives. Now, after some painful experimentation, we have settled on a few Winchester products which fill our needs reliably. We no longer buy, or recommend, any

* A list of second vendors and their phone numbers is given at the end of the document.

‡ It is important when mixing memory module sizes in VAX 11/750s to install the memory in consecutive slots beginning with the first and starting with the 1 Mbyte modules.

† Disk sizes shown throughout this document are in bytes of formatted space available.

removable media disk products.

The choice of available controllers is also wider and much improved. High quality controllers are available which interface to the native busses of 750s and 780s as well as the UNIBUS. In addition, DEC has introduced an entire new storage system architecture which places a great deal more function in the controller, incorporates a new controller-drive interconnect, and uses improved error correction algorithms.

First, we will discuss some of the major areas of change in disk/controller technology. We will then explore how these improve, or otherwise affect, our methods of doing business. Finally, we will consider some specific DEC and non-DEC products.

The availability of large capacity, low cost, high reliability Winchester technology disk drives has had an enormous impact on us. The rack mountable, 300 Megabyte or bigger disk which was always "just around the corner" is really here. It is hard to see how we got along without it. We can now put about 2 Gigabytes of storage in the same footprint that previously held 256 Megabytes. In addition, we consume and dissipate about 25% of the energy we did with older, removable media, drives. The prospective buyer should be warned, however, that not all "winnies" live up to expectations with respect to reliability. We are happy with the reliability of the equipment we describe here. If you want to try something else, be sure and have some long heart to heart talks with other users of the product.

Cost per Megabyte of disk storage is down significantly. Cost ranges from $30 to $110 per Megabyte for disks, not counting the price of the controller(s). This value depends on the size of the units purchased and the choice of vendor. Cost per unit storage in terms of both purchase price and cost to operate are a stronger inverse function of the total drive capacity than ever before. For example, the cost per Megabyte of the 456 Mbyte DEC RA81 is about 35% of that of the 121 Mbyte RA80. The reason for this becomes clear when the drives are examined: many of the components are identical.

The higher recording densities of new disk drives has also been a strong motivator in controller evolution. One technique for increasing the recording density of the drives has been to rely more heavily on sophisticated error correction and block remapping schemes. No large Winchester drive can be depended on to be "error free." In fact, most the drives we use have uncorrectable media defects. These locations must be remapped using some combination of controller firmware and handler software. In addition, the higher bit rates of new disk drives demand faster serial logic in the controller interface. Many older disk controllers are limited to the burst transfer rate of 3330 style disks of about 1.25 Mbyte/sec.

Two types of controller have evolved for the newer, high bit density disks. The first is simply a version of the traditional SMD or Storage Module Drive interface reengineered for higher data rates. This type of interface characterizes all of the non-DEC controllers which have been produced for VAXes of the last few years. These controllers interface to the native busses of the VAX (SBI or CMI) where possible to allow the higher data rates available to be passed all the way through to memory. Where the controller must operate on a bus incapable of a continuous transfer rate as high as the disk, some amount of internal buffering is provided to maximize the amount of date transferred before the disk "blows a rev".*

Non-DEC controllers most often emulate the DEC RH11, RH750, or RH780 interface. Some support for error correction is provided by the controller although a substantial assist is usually required from the system driver. Remapping of uncorrectable media defects is entirely handled by the driver. All 4.2BSD device drivers support bad block remapping. In addition, error correction and remapping support is, optionally, available in the standalone utilities†. The only

---

* By "blowing a rev", we mean a data transfer can not be completed without extraneous disk revolutions. This is mainly a function of the time required by a processor to service an interrupt, the bandwidth of the bus, and the buffering in the controller. With the 4.2BSD file system, disk controllers are now being extended to their limitations, and beyond. This has significantly influenced our concern for the their limitations as bandwidth suffers greatly when such an event takes place.

† Due to limitations in the size of a binary image which may be placed on a boot cassette or floppy, the error

part of the system which does not gracefully handle errors or media defects is the first level bootstrap code used on 750s.

DEC has produced a very different type of controller, partially to deal with the challenges of higher density disk drives. This controller, the UDA50, is an example of DEC's long range plan for mass storage (this "plan" is called the Digital Storage Architecture, or DSA). One of the fundamental goals of DSA is to provide a standard set of disk "operations" across a variety of storage products. With DSA it should be possible to construct standard drivers which know very little about the characteristics or geometry of the actual storage devices they are dealing with. In order to meet this goal, error correction, bad block forwarding, and even the mapping of logical blocks onto the physical disk are handled in the controller. Requests to the controller typically consist of logical block addresses and counts, along with a memory transfer address. Responses then contain either data or a failure message. The controller independently takes all possible measures to recover data before returning failure.

In addition to increasing the functionality of the controller, DSA specifies a new controller to drive interface. The Standard Disk Interface, or SDI, is capable of handling the transfer rates of any drive which DEC may produce in the foreseeable future. This interface is implemented using four electrically isolated radial mini-coax cables to each disk drive embedded in a tough rubber-like umbilical.

On 750 and 780 systems we are, or will be, buying either large (404 Mbyte) Fujitsu disk drives and Emulex SBI or CMI interfaced controllers, or DEC UDA50 controllers with (456 Megabyte) RA81 disk drives. The choice here is not clear as the two packages are both attractive and each has a different set of advantages. Although we do not currently have any UDA50/RA81s at Berkeley, several users of 4BSD do have them, and are very satisfied. In addition, we have visited Colorado Springs, where the drives are manufactured, and run benchmarks on them using an early version of 4.2BSD. The preliminary measurements support our optimism about the UDA50/RA81 combination, though we are not yet ready to publish these results (they will be available at a later time).

It is important not to place too much emphasis on raw performance issues when comparing products as similar in capabilities as the large disk choices presented here. Reliability, freedom from bugs, and ease of maintenance are equally if not more important to us. The value of the product in future configurations is also important. For example, the UDA50/RA81 disk system represents an early implementation of a new architecture. It incorporates many new features heretofore unavailable to us. In addition, it is expandable in the sense that the disk/controller interface is designed to handle future density increases which are not likely to be useable with the traditional SMD interface. On the otherhand, any implementation as new as the UDA50/RA81 is not as likely to be as bug free or as well understood as the traditional RH style interface architecture.

Table 1 indicates some of the tradeoffs as we now understand them.

When searching for less storage for smaller smaller systems, or where two arms are needed for performance and 800+ Megabytes of storage is overkill, another choice is required. Even at $50/Mbyte, a 404 Megabyte drive is not cheap. One of the authors has had good experience on a small 750 system with a 160 Mbyte Winchester disk drive from Tecstore and a National Semiconductor HEX-3000 combination tape and disk controller. We also know of successful use of the Spectra Logic combination controller on a 730 system. Using slightly less expensive disk drives and a combination controller one can obtain cost effective (< $75.00/Mbyte) storage in

---

correction and bad sector forwarding code is not included in the standalone utilities by default.

Criterion	UDA50/RA81	Emulex SC7?0/Fujitsu Eagle
Initial Purchase Cost — 750	UDA50 and 1st RA81 — $57.00/Mbyte w/o additional UNIBUS adaptor; $70.00/Mbyte with UNIBUS adaptor	SC750 and first Eagle — $55.00/Mbyte
Initial Purchase Cost — 780	UDA50 and 1st RA81 — $83.00/Mbyte with UNIBUS adaptor	SC780 and 1st Eagle — $65.00/Mbyte
Cost for Incremental Addition	Additional RA81s — $41.00/Mbyte	Additional Eagles — $32.00/Mbyte
Performance	May be somewhat better in mixed request, multi drive environment due to ordering optimizations possible in controller; software handler at present is suboptimal	Initial tests indicate 5-10% better single file throughput due to better sustained burst rate
Maintenance Costs	Very low — $111/Mo. for 1st drive and controller (compare to $326 for RM05)	Unknown but believed very low
Mean Time Between Failure	Too little experience available yet; RM80 is precursor of RA81 mechanically and has been quite good	Not a lot of experience on these yet either; initial experience looks excellent (smaller Fujis are phenomenal; 30,000 MTBF!)
Mean Time to Repair	Designed for quick field removal of HDA; easy to repair	Not as easy; more complex disassembly
Sources of Maintenance	DEC; maint. contract cheap, real, and available	Not so clear; ask for exchange contract from vendor
Robustness of Drive Interconnect	Incredible — electrical isolation and you could run over cables with a fork lift! Radial connection allows easy removal of a single drive	Same old SMD flat cables; daisy chain
Future Value	Early implementation of new architecture; if it pans out, likely to be compatible with future, high performance, products; DEC resale high anyway	High performance (stretched to limits) implementation of old interface standard; not likely to work again for next increase
Cost to Integrate	Handler is new; some initial bugs likely; probably a bug or two left in controller firmware too	Well known interface; much more likely to be bug free

Table 1. Large Disk System Comparison

smaller amounts and provide a tape interface to boot (so to speak.)

TAPES

We use Emulex TC-11/P UNIBUS tape controllers and Kennedy model 9300-3 800/1600 BPI 125 IPS transports. Cipher tape drives and Wesperco controllers are also widely used. When purchasing second vendor equipment, one will also need cables and a rack in which to mount the tape drive. The Kennedy transport comes with a 15 month factory warranty. Our distributor exchanges/repairs the cards in the controllers based on a local diagnostic mode in the transport. After the warranty period, card swaps cost about $75. For transport mechanical failures the transport is returned to the factory in Monrovia, California, or we fix it ourselves.

George Goble at Purdue is using a 6250 tape system with UNIX. It includes a Telex 6253 drive (800/1600/6250 BPI) 125 IPS with a TELEX Formatter and an Aviv 1 board UNIBUS interface. The UNIBUS interface has 4KB of buffering, to help with bus latency problems, and it really appears to be necessary. The whole system cost him about three times what our 1600 bpi systems cost. The Aviv controller emulates a TU10 which is similar to the Emulex NRZ/PE controller. When heavy data transfer is done to the drive at 6250 bpi it uses the entire bandwidth of the UBA. This forces UNIBUS access through the UBA to be arbitrated by the operating system in order that the tape drive and a disk controller may coexist on the same UBA. **N.B.**: The driver for this controller/transport combination is not currently included in the standard 4BSD system but is trivially cloned from the TM11 handler which is a standard part of the distribution. Aviv also has a TM-11 compatible controller, the TFC 822, which supports both Kennedy and Cipher transports. This controller has more internal buffering than the Emulex TM-11 emulator and may be preferable for this reason.

Name	Speed	Densities
Kennedy	125ips	800/1600
Telex	125ips	800/1600/6250

Our original VAX system came in a package with a DEC TE16 on its own MBA. The TE16 is reliable but slow. The DEC TU45 is faster, but fraught with problems as the high maintenance cost reflects. The DEC TU77 is a good transport, but the auto-loading features do not seem to work well, and it is expensive. Finally, there is a relatively new product from DEC, a 1600/6250bpi 125ips tape drive, the TU78. This is the same transport as the TU77. We have two TU78s in use on campus with mixed results.

The UNIBUS tape drive, the TS11, is included in packages for the 11/750 except for the RK07 package system. It does not have a vacuum column, and is thus hard on tapes. It is a problem to load and has been found to be unreliable.

Name	Speed	Densities	
TS11	45ips	1600	(Not recommended)
TE16	45ips	800/1600	
TU45	75ips	800/1600	(Not recommended)
TU77	125ips	800/1600	
TU78	125ips	1600/6250	

TERMINAL INTERFACES

With a VAX you get 8 lines of DZ-11 that provide some modem control but are not DMA. We use the Able DH-11 emulator, the SuperMAX DH/DM, or one of the two Emulex DH-11 emulators— the CS-11 or CS-21. We also have tried the Intersil DH-11 emulator and know it to function satisfactorily. All of these provide DMA on output and modem control. The CS-11 is unusual in that it provides expansion of up to four 16 line DHs on a single UNIBUS hex module by placing the RS-232 support and UARTS out on the distribution panels and bussing these panels to the UNIBUS module with one ribbon cable. The CS-11 is an attractive solution where a very large number of lines will be connected to one machine since it reduces the number of cables, and UNIBUS backplane space and power required.

4BSD also provide support for the asynchronous serial portion of the the DEC DMF-32. This is the standard communications interface for the VAX 11/730 and has an additional feature of supporting both DMA and programmed interrupt operation for both input and output. The 4BSD driver currently does not use all this flexibility, treating it pretty much like a DH-11. The DMF-32 driver also works with the Able DMZ-11, a product which emulates the asynchronous serial portions of two DMF-32s.

In the area of non-DMA controllers from DEC, there are the DZ-11 and DZ-32 (a DZ-11 with full modem control).

Both the DZ's and the DH's have input silo's that UNIX can use to reduce interrupt load on input. The DMA output of the DH emulators is especially important for graphics applications where high-volume and continuous output occurs.

PRINTERS

One of the most exciting developments in the area of printers is the availability of desk top laser printers. This paper was printed on an Imagen laser printer we have been using, quite successfully, for several months now. The Imagen offers high resolution (240 dots/inch), uses plain paper, and seems to require minimal hardware maintenance. It is interfaced to one of our VAXes via a 19.2 Kbaud RS-232 line although a parallel interface is also available.

Among the problems with the Imagen are the small number of available fonts and the incompleteness of some of those which are available. In addition, the Cannon LBP-10 printing engine used has only a 200 sheet paper tray. Since the unit employs a wet process Xerography and smells a bit, it is not located in the same room as a person who might be responsible for refilling the tray. This inevitably results in print jobs backing up in a long queue until someone notices paper is needed. The Imagen folks were initially TEX oriented and their *troff* support contains glitches which are purported to go away with future releases of the software. We also hope to eventually interface our printer directly to the Ethernet; as soon as Imagen provides the necessary software to do so.

Another laser printer based on the Canon LBP-10 engine is produced by Symbolics. Symbolics offers both RS-232 and parallel interfaces to the printer. The Symbolics software is known to provide excellent software support for *troff*. We are are now evaluating a Symbolics printer.

QMS in Georgia has apparently solved the mysteries of the Xerox 2700 printer and is distributing an OEM version which might be a good choice. The major potential advantages here have to do with Xerox's size and extensive field support. The unit is dry process (unlike the Imagen and Symbolics) and has 300 dots/inch resolution. With any luck, we will also be evaluating this unit soon.

We have been using some Printronix 300 and 600 line per minute dot-matrix printers. The Printronix printers do point-plotting at 60 points per inch. They are not outstandingly cheap, but are ruggedly built.

The new Data Products B-600-1 is a 600 LPM band printer. We have one and are buying another. Although we had some initial problems getting the first unit into service, it now runs reliably and is our heaviest usage production printer.

PLOTTERS

Electrostatic printer/plotters that are capable of 200 dots/inch are usable both as plotters and as output devices for *troff*. We have an old model Varian that requires considerable care and feeding; newer models are said to be less of a headache. A new Versatec 11" model sells for about $8,000. The objections to all these guys are that the paper tends to be wet sometimes, stinky, and more expensive than line printer ($20 per 1000 sheets). These are high maintenance items as are all heavily used hardcopy output devices we are familiar with. For *troff*, we now vastly prefer the Imagen laser printer mentioned above.

NETWORK INTERFACES

Networks can be categorized as *local area networks (LANs)* or *long haul networks* according to their geographical limitations. The most widely publicized local area network is the Ethernet. An example of a long haul network is the DARPA Internet which spans many continents and includes devices such as communication satellites for connecting disjoint *sub-networks*.

Among local area networks there are several competing modulation schemes. The Ethernet and several other networks uses *baseband* modulation techniques, while newer technologies, such as *broadband*, are available from other vendors. Some of the major differences between baseband and broadband technologies are maximum station separation, cable bandwidth, and, currently, per station connection cost. At this time, the least expensive, and most readily available local area networking hardware use baseband modulation. However, given the limitations inherent in baseband modulation schemes, companies are placing more work into developing low cost parts for use in broadband networks.

Aside from the question of baseband versus broadband, selection of medium is an issue. Coax cable is commonly used but types of coax vary. Broadband networks normally use the same standard 75 ohm coaxial cable used for CATV, while baseband uses 50 ohm cable. This implies that upgrading a network from baseband to broadband requires expensive installation of a new cable unless one thinks ahead, or your site already has installed cabling for in-house CATV use. Further, the best medium in terms of signal loss and noise immunity is fiber optic cable. However, due to problems such as tapping the cable, few vendors have selected this technology. If you plan to consider broadband at some time in the future, while at the outset using baseband, it is well worth the cost of the extra cable to run parallel 50 and 75 ohm coax.

In looking at network controllers, we will consider only the available local area networking hardware; our experience with long haul networks is limited to the Internet and so is of minimal interest.

There are at least four vendors with existing or announced Ethernet controllers, and with the soon to be available "Ethernet chips" more vendors may announce products. It is unlikely, however, that the Ethernet chips will significantly influence the current prices as the price of an Ethernet controller has already been driven down by the market competition. While the influx of new technology may not lower controller prices, it is sure to improve their performance and reliability.

We currently use 10Mb/s UNIBUS Ethernet controllers from both Interlan and 3Com. The two controllers have almost identical throughput characteristics with 4.2BSD, but neither have proven entirely satisfactory. The 3Com controller is the less expensive of the two. Its design is optimal for small PDP-11s and LSI-11s where the processor is resident on the same bus with the controller. The design employs 16 or 32 Kbytes of dual-ported RAM which is directly addressable as UNIBUS (or Q-bus) memory. While this is effective for machines such as the PDP-11 or LSI-11 where no penalty is required when accessing the on-board memory, with a VAX any memory access must be arbitrated by the intervening UNIBUS adaptor. The result of this is that accesses to the on-board memory are heavily constrained by the characteristics of the UNIBUS adaptor.

In accessing memory through a UNIBUS adaptor, all accesses must be performed on even byte boundaries and be no more than two bytes at a time. Consequently, one must either be very careful about the coding of a network interface driver, or the contents of any on-board memory must be copied into main memory before manipulating it. Due to the architecture of the networking subsystem included in 4.2BSD and the lack of control over the code generated by the VAX C compiler, constraining memory fetches was infeasible and the second alternative was taken. This implies that data must be block copied in to and out of the on-board memory a word at a time. The VAX *movc3* instruction is not usable in the UNIBUS address space, making this an expensive operation.

A second problem with the 3Com controller is that it lacks an on-board timer for implementing a backoff algorithm when accessing the Ethernet. This implies the host must perform a timing loop when backing off from a congested Ethernet. When an Ethernet is heavily congested this may prove to be very costly as no other processing may take place while the host timing loop is executing.

A third problem with the 3Com controller is that it does not allow a host to receive its own broadcast packets. This implies that broadcast packets must be captured in software. We consider this a serious deficiency as it prevents hardware testing without an auxiliary echo server.

The second Ethernet controller we have used is made by Interlan. This controller provides DMA access, as well as several desirable features such as on-board retransmissions. Unfortunately, while the DMA interface should be expected to provide higher throughput than the shared memory approach, using the Interlan interface we have been able to attain only comparable transfer rates to those measured with the 3Com interface. In addition, the controller consumes a significant amount of of +5 volt power. While broadcast packets are retrieved by the interface, the Ethernet CRC calculation is not performed.

We know of two other Ethernet controllers, one from ACC and one from DEC. We have two ACC controllers for evaluation, but have yet to gain any experience with them. The ACC controller is based on the UMC-Z80 and provides a DMA host interface. The DEC Ethernet controller was announced at the last DECUS meeting, but as of yet we know of none in customer hands.

To summarize the Ethernet controller situation, it appears the best strategy to follow is to wait for Ethernet chips to become widely available so the vendors can reengineer their existing controllers with minimal cost. If you require Ethernet access from your VAX now, you may wish to follow our approach: select the lowest priced product and treat it as "disposable" in the expectation that something better will eventually be available.

Other than Ethernet, the Proteon proNET 10 Mb/s ring network is also popular. This device is also known as the Version II lni ring network and is in heavy use at LBL and MIT with good results. The Proteon proNET outperforms both the 3Com and Interlan controllers mentioned above in throughput benchmarks run with the 4.2BSD networking support. Further, the ring design eliminates the standard complaints about ring architectures by use of a star-shaped ring configuration. The star-shaped ring allows easy addition and deletion of nodes without splicing drilling or taping. Also, any node can fail without bringing down the ring because it is bypassed at the star-shaped ring's passive wire center. The major concern with a ring network is that it is

incompatible with the de facto standard Ethernet. Cost per station is slightly higher than the Ethernet, but startup costs are lower (unless you use a fiber optic wire center). Proteon has announced they are working on an 80 Mb/s controller which should make the network even more attractive.

SOFTWARE SUPPORT

There has been increasing demand for 4BSD at commercial installations in a form less expensive and more digestible than a source license from Western Electric and an unsupported distribution from Berkeley. A number of companies, licensed by Western Electric to sell and support UNIX in binary form, are now distributing 4BSD. Some of these companies support 4BSD as an enhancement for their hardware offerings others deal only in software. Licenses from these vendors normally cost much less than a UNIX source code license. These companies usually try to make 4BSD more palatable to the non-academic community by providing more first-time user documentation and specialized consulting addressing specific customer applications. More formal software support arrangements than those offered by U. C. Berkeley are also available. 4BSD software sales and support vendors are included in the list at the end of this paper.

SYSTEM PACKAGES

We now present some sample system packages. Each represents a balanced system for timesharing use under UNIX. People often ask us how many users can be supported UNIX in these configurations. In the absence of specific information about applications to be run, this is an unanswerable question. The amount of load presented to the system by different applications varies widely. We mention with each system the count of interactive users typically supported in our university research environment.

We first present systems based on 11/750s and then systems based on 11/780s. With each example we suggest functionally similar systems configured in at least two different ways: first with as much equipment as possible from DEC and second with the best equipment known to us. We will not consider the VAX 11/730 as we believe it is not a viable option for most timesharing environments. Our experience with the 730 indicates it has approximately the raw processing power of a PDP-11/34 size CPU. Thus, even though it is a reasonable choice for people looking for an entry level VAX, we consider it mostly a single user machine.

Various measurements of the speed of the 11/750 and 11/780 indicate that the 11/750 executes at roughly 60 percent of the speed of an 11/780. By comparison, an 11/70 runs at roughly 75 percent of the speed of an 11/780 using the same benchmarks, which involve no floating point, no 32 bit arithmetic on the 11/70, and no system calls. For UNIX time sharing usage we believe that the 11/750 has better performance than an 11/70. This is due mainly to additional tuning and performance enhancements to the VAX kernel, and to the larger address space of the VAX architecture.

The first system we consider is a small 11/750. This is followed by an expansion of the 11/750 into a larger system. We are fond of the VAX 11/750 as it provides the most computational power per unit cost of the three VAX implementations.

The second base system is a small 11/780. We show how it can be built from a DEC RUA81/TU78 package system, and how to build it from mixed vendor equipment. We then expand it in two increments.

The small systems we suggest start with a single disk and tape controller and some memory. For time-sharing applications we configure our VAX systems allowing 256K bytes of memory for the kernel and roughly an additional 100k bytes of memory per active user.* Memory is cheap, especially for the 11/780, so we don't skimp on it.

With more than a few users, it is critical that more than one disk arm be present in the system. Thus all but the smallest systems include more than one disk. As the active user count rises, having more than one disk controller is also a good idea. The large system packages include two disk controllers. For really large and i/o intensive systems we recommend high bit density disk drives like the Fujitsu Eagle or the RP07 drive from DEC as they provide a higher transfer rate than the 1.25 Mbytes typical of the remaining drives. Using this transfer rate effectively requires running with interleaved memory.

It is desirable on all UNIX systems to have at least 100MB of disk space so that all the source for the system and all the standard programs may be kept on line with some room for locally developed programs. The amount of space required by user programs varies per installation; we manage to run many of our instructional/research machines using about 300-600 megabytes of space actively, although slightly more than this would be desirable.

Our large research machine runs with 1 Gigabyte of disk storage, with 2 disks on a UNIBUS and 2 disks on MASSBUS adapters. The weakest point in this system is that it has only a 45ips TE16 tape drive for backups. For even the smallest systems, 45ips will soon seem slow. We therefore recommend starting with a 125ips 1600bpi tape drive. As full 2400 foot tape reels

* These numbers work reasonably well in an environment typical of University work (course work, paper preparation, debugging programs, developing applications for microcomputers, etc.) More demanding applications could require substantially more memory per user.

hold only 30MB at 1600bpi, large systems should consider including at least one tape drive capable of writing 6250bpi tapes.

VAX 11/750 PACKAGES

We want to put together a small 11/750 system capable of supporting about 8 time-sharing UNIX users, and a larger 11/750 system for roughly 16-24 users. We need a minimum of 100 megabytes of space for the small system and a reasonable tape drive, preferably a 125ips unit so that tape operations can be done in a reasonable amount of time; if the system is to include only non-removable disks, we consider the faster tape system to be important. For the larger system, we wish to add disk space to give the system a minimum of 250 megabytes of space, and have more than one disk arm.

Small system

Small 750 System		
	DEC System	Mixed Vendor System
CPU	11/750	11/750 from Broker or Integrator with .50 Mbyte DEC Memory but 8 Mbyte capacity.
Memory	1 Mbyte DEC	1 Mbyte National/Trendata/Mostek
Disk System	UDA50 Unibus Controller RA80 121 Mbyte Drive	Emulex SC750 RH750 Emulator Fujitsu 134 Mbyte Drive
Tape System	TGE16 45 ips Tape Sys.	Emulex or Wesperco Controller Cipher or Kennedy 125 ips tape

The small DEC system is based on the SV-BXGMB-CA package, and includes an RL02 in addition to the RA80. We basically ignore the RL02 which is of little use to us and use the package because it is the cheapest way to get started. We add a TGE16 tape system as the best choice among a myriad of evils. It is really too slow, but it is reliable and not too expensive. DEC has been promising some better low cost tape units soon.

The mixed vendor system is as inexpensive as possible while retaining upward expandability. If the builder were sure that this system was not going to be expanded much then a substantial amount more could be shaved from the cost by making several substitutions. A National Semiconductor or Spectra Logics UNIBUS combination disk and tape controller could be substituted for the separate CMI disk controller and UNIBUS tape controller shown. A slower, perhaps 45 ips, tape unit with built in formatter could be substituted for the 125 ips tape drive. An older CPU with 2 Megabyte maximum memory capacity could be used. These are available for substantially less than the CPUs equipped with the newer memory controller and backplane. Even with these modifications, another disk and another Megabyte of memory could easily be added to produce substantial performance improvement. One advantage of the mixed vendor system as shown is that the Emulex SC750 controller keeps the disk drives off the UNIBUS. If an Ethernet controllers is added to the system, they will not be contending for the bus.

Medium system.

To expand this basic system to support more users, we would add additional lines, disk storage and memory. To the small all-DEC system we would add another RA80, another Megabyte of memory and a DZ-11E. To the mixed vendor system we would add another Fujitsu 134 Mbyte

disk, an Able or Emulex DH-11 emulator and another Mbyte of memory:

Augmenting the Small 750 to a "Medium" System		
	DEC System	Mixed Vendor System
Additional Disk	RA80 121 Mbyte Drive	Fujitsu 134 Mbyte Drive
More Memory	1 Mbyte DEC	1 Mbyte National/Trendata/Mostek
More Serial Lines	DZ-11E	Able/Emulex "DH"

There are, of course, further expansion possibilities for the 11/750. These vary depending on the application but could include a floating point accelerator, more memory up to 8 Mbytes, and an additional UNIBUS adaptor on the DEC system if other high speed devices like network interfaces are to be on the UNIBUS along with the UDA50.

VAX 11/780 PACKAGES

For a system with more growth possibilities than an 11/750, faster processing, and higher i/o bandwidth, we recommend starting with a small 11/780. Our goal here is to start with a system capable of supporting 8-16 timesharing users and expanding the system to be capable of supporting roughly 24 users. We also consider a large expansion of this system, to a system that might support 32 to 40 terminal users to the exhaustion of available CPU cycles.*

Small system

For our small system we use 400 Megabytes of disk storage and a 125ips 6250bpi tape drive that will be capable of handling file backups if the system is eventually expanded. In our first expansion of this small system, we wish to add to the available space to a minimum of 800 Megabytes of disk storage, acquire at least two disk arms, and add additional terminal lines. In a large expansion of this system we include more terminals, an additional disk controller to get at least two separate disk channels, and an additional 800 Megabytes of storage for a total of 1600 Megabytes.

To build a small system from all DEC equipment, we would start with the RUA81/TU78 based system, the SV-AXECA-CA. This system includes 8 terminal lines, 4 Megabytes of memory, a 456 Megabyte disk drive and a 125ips 6250bpi tape. The system is equipped with two UNIBUS adaptors so that the UDA50 does not contend with other UNIBUS devices. To this we would add a floating point accelerator.

On the mixed vendor system we would substitute a Fujitsu Eagle 404 Mbyte disk drive on an Emulex SC780 SBI interfaced controller and an Aviv/Telex 6250 tape subsystem.

Small 780 System		
	DEC System	Mixed Vendor System
CPU	11/780	11/780 from Broker or Integrator with .25 Mbyte DEC Memory and UNIBUS Adaptor Included
Memory	4 Mbyte DEC	4 Mbyte National/Trendata/Mostek
Disk System	UDA50 Unibus Controller RA81 456 Mbyte Drive on own UBA	Emulex SC780 RH780 Emulator Fujitsu 404 Mbyte Drive
Tape System	TEU78 125ips 6250 ips Tape Subsystem	Aviv Controller Telex Drive/Formatter
Serial Lines	DZ-11A	Able/Emulex "DH"
Other	DEC Floating Pt. Acc.	DEC Floating Pt. Acc.

Medium system

To expand this basic system to support more users and get additional disk space, we would add additional lines and disk storage.

* Using systems similar to the largest shown here, in an environment consisting of small student programming some sites have reported running up to 70 interactive users; CPU cycles are the critical resource with this many users.

Augmenting the Small 780 to a "Medium" System		
	DEC System	Mixed Vendor System
Additional Disk	RA81 456 Mbyte Drive	Fujitsu 404 Mbyte Drive
More Serial Lines	DZ-11E	Able/Emulex "DH"

Large system

To form a system with the emphasis on handling of data-intensive applications, and to emphasize total growth of the system, we would add a second disk channel and interleave memory to increase i/o throughput and reduce average CPU memory access as much as possible. In both the DEC and mixed vendor systems a CPU extension cabinet would be required in addition to another DEC memory controller. We would fill out the second memory system to 4 Megabytes.

For more disk throughput, we would add an REP07-AA 504MB disk drive on a MASSBUS controller to the basic DEC system. This disk provides a very high burst data throughput and could share the MASSBUS Adaptor of the Tape Unit with only minor performance loss while the tape unit was being used.

To accomplish the same ends with the mixed vendor system, we would simply add a second Emulex SC780 disk controller channel and at least one more Fujitsu Eagle 404 Mbyte disk drive.

Augmenting "Medium" 780 to "Really Big" System		
	DEC System	Mixed Vendor System
Additional Disk and Channel	RP07 (516 Mbyte) on MASSBUS with tape sys.	Fujitsu Eagle (404 Mbyte) on another SC780 controller
Second Memory Controller and Cabinet	DEC	DEC
Additional Memory	DEC	Trendata/National Mostek
More Serial Lines	DZ-11E	Able/Emulex "DH"

SUGGESTIONS ON BUYING HARDWARE

The are a variety of ways in which you can acquire the systems we have suggested here, whether they be all DEC or mixed vendor. Your choice of acquisition methods depends on a number of factors including:

- How much can you afford to pay?
- How long can you wait?
- How much risk and responsibility are you willing to assume for integrating your own hardware components?
- What kind of maintenance is available to you?
- How much help you need in integrating 4BSD?

Here is a simplified breakdown of the possibilities:

1. **Buy as much as possible from your DEC marketing organization.**

   Although this solution, in our experience, takes the longest and costs the most, it has its advantages. DEC is likely to ship you a well tested, integrated system, close to the time initially promised. In most cases they will support you well through any initial start-up problems with the hardware. The system bought this way will automatically be accepted for a DEC maintenance contract. Of course, the can't help you much with 4BSD (yet). Also, they are not likely to be very flexible about adjusting their configuration to your needs.

2. **Buy an all-DEC system from a an OEM specializing in 4BSD**

   These OEMs are a relatively new phenomenon. They usually get a much better discount from DEC on hardware and can pass part of this through to you in terms of UNIX expertise as well as reduced cost. Sometimes they will be able to deliver hardware quickly when DEC is telling you months. Since they sell largely DEC systems, you can still take advantage of DEC Field Service and most systems sold this way are guaranteed acceptable for a DEC maintenance contract.

3. **Buy a mixed vendor system from a systems integrator**

   DEC has had a long love/hate relationship with people who specialize in building systems which use DEC's CPUs and other manufacturers peripherals. We think these integrators serve many useful functions. First, and foremost, they often build a cheaper and better system, frequently on short notice. Second, they keep DEC honest. Sometimes we feel they should charge for their quotations, since these are often used advantageously to encourage DEC to come down to a more reasonable price on a system.

   Don't assume mixed vendor systems are not maintainable. There is a whole spectrum of maintenance possibilities for these systems, particularly in major metropolitan areas. If you are considering this route, be sure and spend some time on the phone with the customers of your prospective vendor. Insist on the names of *long term* customers, and talk a lot about maintenance experience. The folks we mention on the last page of this paper are known to have experience with 4BSD.

4. **Integrate the mixed vendor system yourself**

   If you are qualified for this adventure, then you probably know who you are. We can't begin to tell you all the pitfalls. Start small. Buy a mostly integrated system and add something you can afford to have not work for a while, such as more memory (almost too easy), or a better tape drive, or more terminal interfaces. If you really want to do the whole thing, finding the CPU is one of the harder parts. Get yourself a copy of *Computer Hot Line*. You can probably get a complimentary copy by calling them at (800) 247-2244. This is the social register of computer brokers and a substantial portion is dedicated to folks selling new and used DEC. (Hot Line, Inc. also distributes the Farm Machinery Hot Line and several other classified flea market variety publications. They can not be expected to control the content of adds. Use at your own risk!)

We would like to make two more observations about buying equipment. It has been our experience that the service you will receive from your source is directly proportional to the risk in using that source. Further, the service often is inversely proportional to the sources size. Loosely translated, little guys work harder.

Many who have dealt with DEC sales report disappointing experiences. Lack of product knowledge and inability to bend to customer needs are typical complaints. This is not to say that there are not excellent DEC sales people. There are. And you must remember, when you finally close that deal with your DEC salesperson, **it will be delivered,** eventually.

On the other hand, the systems integrator who builds one or two systems a month typically succeeds or fails based of the experiences of his small customer base. We have known many of these folks to make superhuman efforts to pull together a customer system, often succeeding

without half the resources available to DEC sales people. They are also much quicker to pick up trends like an interest in 4BSD and start to mold their services to fit. Once again, there is always the exception, the "Unix Systems Integrator" who couldn't tell an inode from a tree toad. If you go this route, you have a good selection to choose from. **Spend time talking to previous customers.**

CONCLUSIONS

We have presented sample VAX systems over a wide performance range using both all-DEC and the best available second vendor equipment, emphasizing, independently, minimal cost and maximal expandability. Use this information wisely; price shouldn't always be the bottom line.

Consider the all-DEC system if you can afford it. If not, the second-vendor equipment in the packages here is all thought to work well on VAX hardware. You can reliably build and operate such a system. When you have struggled through your particular set of difficulties and are up and on the uucp network, be sure and write us about your experiences. Good luck!

ACKNOWLEDGEMENTS

This document represents a lot of work. It would have been easier, except for everyone who sent us helpful hints and suggestions and, in general, kept us honest. In particular, we would like to acknowledge all those vendors who were patient with us, especially those whose products were ultimately not included. George Goble at Purdue made several helpful comments which greatly improved the content of the document, and his experiences with Fujitsu Eagles has made a significant impression on us. The DEC DSA engineering team in Colorado Springs, including Paul Massiglia, Bill Grace and Chuck Hess were particularly generous with their time and energies. Peter Weinberger of Bell Laboratories shared his experiences with the UDA50/RA81 with us. Kirk McKusick spent time traveling to Colorado Springs to aid in evaluating the DEC RA81 disk drive. David Mosher has worked diligently as the purchasing agent for CSRG and also contributed to our understanding of the subtler points of disk manufacturing and operation. Jim Reeds gave the paper a careful proof reading and found many oversights.

VENDOR REFERENCES

Manufacturer	Product	Phone	Vendor contact
3Com	Ethernet Cont.	(415) 961-9602	3Com (Mike Hallaburka)
Able	Async. Mux	(714) 979-7030	Able Computer (Norm Kiefer)
Aviv	Tape controllers	(619) 247-6844	Aviv (Ed Hagenbuch)
Data Products	Printers	(415) 948-8961	MQI Associates (Avery Blake)
Emulex	Controllers	(415) 820-2933	Eakins Associates (Bob Sigal)
Fujitsu	Disks	(415) 969-5109	Eakins Associates (Bob Sigal)
Imagen	Laser Printers	(415) 960-0714	Imagen (Bob Wallace)
Interlan	Ethernet Cont.	(714) 752-4002	Interlan (Gary Steadman)
Intersil	Async. Mux	(408) 743-4300	Intersil (Alan Truscott)
Kennedy	Tape Transports	(408) 245-9291	Electronic Marketing Specialists
Mt Xinu	4BSD Binary Sales	(415) 644-0146	MtXinu (Bob Kridle)
National	Memory	(800) 538-8514	National (Don Johnson)
National	Disk/Tape Cont.	(800) 538-8514	National (Don Rudolph)
NMS	Disk/Tape Sys	(415) 443-1669	NMS(Bob Crippen)
Printronix	Printers	(408) 245-4392	Group III Elect. (Scott Drzewiecki)
Proteon	Network Cont.	(617) 894-1980	Proteon (Al Marshall)
Spectralogics	Disk/Tape Sys.	(415) 443-1669	Nat. Mem. Sys. (Bob Crippen)
Symbolics	Laser Printer	(415) 494-8081	Symbolics (David Shlager)
Tecstore	Disks	(408) 732-2143	Tecstore (Mel Feintuch)
Trendata	Memory	(714) 540-3605	Trendata (Miles Efron)
Varian	Plotters	(408) 733-2900	Varian (Ted Downs)
Versatec	Plotters	(415) 828-6610	Versatec (Bruce Fihe)

SYSTEM INTEGRATION/SUPPORT

Name	Phone	Contact	Notes
VLSI	(415) 490-3555	Joe Voelker	Mixed Vendor Systems and Support
IDS	(408) 738-3368	Dick Cavanaugh	Specialize in All DEC Systems
Eakins Assoc.	(415) 969-4533	Bob Sigal	Mixed Vendor Systems and Support
IPS	(713) 776-0071		Mixed Vendor Systems
Iverson Inc.	(415) 459-5665	Jon Iverson	Mixed Vendor Integration
UNIQ	(415) 362-0470		All DEC Systems

MORE/bsd Volume VI

System Management
Mail
Networking and Communications

Installing and Operating 4.2BSD on the VAX
Building 4.2BSD UNIX Systems with Config
4.2BSD System Manual
Hints on Configuring VAX Systems for UNIX
Disc Quotas in a UNIX Environment
4.2BSD Line Printer Spooler Manual
Fsck — The UNIX File System Check Program
On the Security of UNIX
Password Security: A Case History
The UNIX Time-Sharing System
UNIX Implementation
The UNIX I/O System
Bug fixes and changes in 4.2BSD
A Fast File System for UNIX
Mail Reference Manual
A Dial-Up Network of UNIX Systems
Uucp Implementation Description
SENDMAIL — An Internetwork Mail Router
SENDMAIL Installation and Operation Guide
4.2BSD Networking Implementation Notes
A 4.2BSD Interprocess Communication Primer

MORE/bsd Volume III

UNIX Shells
Tools and Utilities
Ingres

An Introduction to the C Shell
An Introduction to the UNIX Shell
UNIX Programming — Second Edition
Make — A Program for Maintaining Computer Programs
A Tutorial Introduction to ADB
Yacc — Yet Another Compiler-Compiler
Lex — A Lexical Analyzer Generator
Screen Updating and Cursor Movement Optimization: A Library Package
The M4 Macro Processor
SED — A Non-interactive Text Editor
Awk — A Pattern Scanning and Processing Language (Second Edition)
DC — An Interactive Desk Calculator
BC — An Arbitrary Precision Desk-Calculator Language
An Introduction to the Source Code Control System
Source Code Control System User's Guide
LEARN — Computer-Aided Instruction on UNIX (Second Edition)
A Guide to the Dungeons of Doom
INGRES Version 7 Reference Manual

MORE/bsd Volume V

Editing
Document Preparation

An Introduction to Display Editing with Vi
Edit: A Tutorial
Ex Reference Manual
Ex Changes — Version 3.1 to 3.5
A Tutorial Introduction to the UNIX Text Editor
Advanced Editing on UNIX
NROFF/TROFF User's Manual
A TROFF Tutorial
Writing Papers with NROFF Using —me
—me Reference Manual
Typing Documents on the UNIX System:
Using the —ms Macros with Troff and Nroff
A Revised Version of —ms
Tbl — A Program to Format Tables
Typesetting Mathematics — User's Guide (Second Edition)
A System for Typesetting Mathematics
Writing Tools — The STYLE and DICTION Programs
Updating Publication Lists
Some Applications of Inverted Indexes on the UNIX System
Refer — A Bibliography System
Berkeley Font Catalog

Berkeley Font Catalog

October 1980

Introduction

This catalog gives samples of the various fonts available at Berkeley using vtroff on our Versatec and Varian. We have them working 4 pages across in a 36 inch Versatec, and rotated 90 degrees on a Benson-Varian 11 inch plotter. The same software should be adaptable to an 11 inch Versatec, and in fact is running at several other sites, however, not having one here, it isn't part of this distribution. Such a driver is available from Tom Ferrin at UCSF.

To use these fonts:

(1) Hershey. This is the default font. The Hershey font is currently the *only* complete font, with all 16 point sizes and all the special characters troff knows about. To get it, use vtroff directly. To illustrate this with the -ms macro package:

vtroff -ms paper.nr

(2) Fonts with roman, italic, and bold, such as nonie. You can load all three fonts with, for example:

vtroff -F nonie -ms paper.nr

To get just one of these fonts, use (3) below, appending .r, .i, or .b to the font name to specify which font you want mounted, e.g., to get italics in delegate,

vtroff -2 delegate.i -ms paper.nr

(3) To get a font without a complete set, choose which font (1, 2, or 3) you want replaced by the chosen font. For example, to use bocklin as though it were bold, since font 3 is bold, use:

vtroff -3 bocklin -ms paper.nr

To switch between fonts in troff, use

.ft 3

to switch to font 3, for example, or use

\f3word\f1

to switch within a line. For more information see the Nroff/Troff Users Manual.

Special note: *troff* thinks it is talking to a CAT phototypesetter. Thus, it does all sorts of strange things, such as enforcing restrictions like 7.54 inches maximum width, 4 fonts, a certain 16 point sizes, proportional spacing by point size, etc.

In particular, the following glyphs will *always* be taken from the special font, no matter what font you are using at the time:

@, #, ", ', `, <, >, \, {, }, ~, ^, and _

This may explain what are otherwise surprising results in some of the subsequent pages.

In addition, the following Greek letters have been decreed by *troff* as looking so much like their Roman counterparts that the Roman version (font 1) is always printed, no matter what font is mounted on font 1 at the time:

A, B, E, Z, H, I, K, M, N, O, P, T, X.

(See table II in the back of the Nroff/Troff Users's Manual for details about what glyphs are in each font and how to generate the special glyphs.)

Font Layout Positions

Code	Normal		Special		Code	Normal	Special	
000					100			
001	fi	\(fi	■	\(if	101	A	A	\(*A
002	fl	\(fl	▭	\(ip	102	B	B	\(*B
003	ff	\(ff	▬	\(pt	103	C	Γ	\(*G
004	—	\(-	▮	\(rh	104	D	Δ	\(*D
005	–	\(ru	∪	\(cu	105	E	E	\(*E
006	—	\(em	∩	\(rn	106	F	Z	\(*Z
007	•	\(bu	⊗	\(bs	107	G	H	\(*Y
010	•	\(sq	±	\(+-	110	H	Θ	\(*H
011		\(fl	≤	\(<=	111	I	I	\(*I
012		\(fL	≥	\(>=	112	J	K	\(*K
013	•	\(de	√	\(sr	113	K	Λ	\(*L
014	†	\(dg	∽	\(ts	114	L	M	\(*M
015	'	\(fm	∫	\(is	115	M	N	\(*N
016	●	\(co	∕	\(sl	116	N	Ξ	\(*C
017	●	\(rg	|	\(bv	117	O	O	\(*O
020	♦	\(ct	⌊	\(lf	120	P	Π	\(*P
021	¼	\(14	⌋	\(rf	121	Q	P	\(*R
022	½	\(12	⌈	\(lc	122	R	Σ	\(*S
023	¾	\(34	⌉	\(rc	123	S	T	\(*T
024			⎧	\(lt	124	T	Υ	\(*U
025			⎩	\(lb	125	U	Φ	\(*F
026			⎫	\(rt	126	V	X	\(*X
027			⎭	\(rb	127	W	Ψ	\(*Q
030			⎨	\(lk	130	X	Ω	\(*W
031			⎬	\(rk	131	Y	∓	\(dd
032			⊂	\(sb	132	Z	⌊	\(br
033			⊃	\(sp	133	[	⌈	\(ib
034			∩	\(ca	134		∕	\e
035			∈	\(no	135	]	∅	\(ci
036			∉	\(lh	136		∂	
037			∈	\(mo	137	_	∞	_
040	space				140		′	′
041	!				141	a	α	\(*a
042			∀		142	b	β	\(*b
043			#		143	c	γ	\(*g
044	$				144	d	δ	\(*d
045	%				145	e	ε	\(*e
046	&				146	f	ζ	\(*z
047			∍		147	g	η	\(*y
050	(		∇	\(gr	150	h	θ	\(*h
051	)				151	i	ι	\(*i
052	*		×	\(mu	152	j	κ	\(*k
053	+		+	\(pl	153	k	λ	\(*l
054	,				154	l	μ	\(*m
055	-		−	\(mi	155	m	ν	\(*n
056	.				156	n	ξ	\(*c
057	/		÷	\(di	157	o	ο	\(*o
060	0		≡	\(==	160	p	π	\(*p
061	1		≅	\(~=	161	q	ρ	\(*r
062	2		≈	\(ap	162	r	σ	\(*s
063	3		≠	\(!=	163	s	τ	\(*t
064	4		←	\(<-	164	t	υ	\(*u
065	5		→	\(->	165	u	ϖ	\(*f
066	6		↑	\(ua	166	v	χ	\(*x
067	7		↓	\(da	167	w	ψ	\(*q
070	8		§	\(sc	170	x	ω	\(*w
071	9		°	\(**	171	y	∂	\(pd
072	:				172	z	∅	\(es
073	;				173		{	{
074			<		174	|	|	\(or
075			=		175		}	}
076			>		176		~	~
077	?				177			

APL FONT, 10 POINT ONLY

A⍺B⊥C∩D⊂E∈ F⌷G⍫H⍋I⍳J⍤K' L⌹M⌈N⍱O○P Q?R⍴S⌊T~U↓ V∪W X⊃Y↑Z⊂ 01234 56789*

(" # $ = × ≥ ∨ ∧ ≤ ≠ ÷ ¨ { } { } ⍲ ~ ⍁ \ ∃ ⍟ → < ÷ / \ . > , <

*! → (⍰ → = & → × ′ → ′ (→ ∨) → ∧ : → ≤ * → ≠ - → ÷ = → ¨ [→ {] → } | → ∃ ' → → ; → < ÷ → ÷ ? → *

Baskerville font, roman, ibold, italic, 12 point only (Called "basker" on line.)

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; ÷ / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; ÷ / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; ÷ / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Bocklin font, 14 and 28 point only.

14 point

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

" ' ( ) : - = [ ] ' ; / ? . ,

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality: since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

28 point (No punctuation except period.)

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789 .

If time be of all things the most precious wasting time must be as Poor Richard says the greatest prodigality since as he elsewhere tells us lost time is never found again and what we call time enough always proves little enough Let us then up and be doing and doing to the purpose so by diligence shall we do more with less perplexity.

Bodoni font, roman, bold, *italic*, 10 point only.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ‘ ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ‘ ; + / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ‘ ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Chess, 18 point only

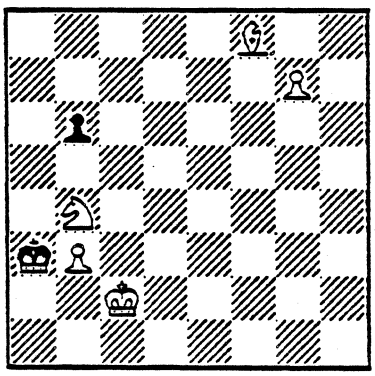
Note: Our attempt at compatibility with Stanford was only 99% successful. If you use a blank space to indicate an empty white square it will come out narrow due to the stupidity of troff. Either include the line

```
.cs ch 36
```

to put yourself in constant spacing mode or else use zero instead of space. You should also set the vertical spacing to 18 points.

```
.nf
.ft ch
.cs ch 36
.ps 18
.vs 18
HITTTTTTTX
V0Z0Z0A0ZF
VZ0Z0Z0O0F
V0o0Z0Z0ZF
VZ0Z0Z0Z0F
V0M0Z0Z0ZF
VjPZ0Z0Z0F
V0ZKZ0Z0ZF
VZ0Z0Z0Z0F
WUUUUUUUUG
.sp
.ft P
.ps 8
.cs P
```

p	♟	P	♙
o	♟	O	♙
b	♝	B	♗
a	♝	A	♗
n	♞	N	♘
m	♞	M	♘
r	♜	R	♖
s	♜	S	♖
q	♛	Q	♕
l	♛	L	♕
k	♚	K	♔
j	♚	J	♔
U	—	T	—
F	|	V	|
G	.	W	.
X	.	H	.
0		Z	



White mates in three moves.

Clarendon, 14 and 18 point roman only. From SAIL (Paul Martin & Andy Moorer)

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

" # $ % × ' ( ) : - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

" # $ % × ' ( ) : - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Computer Modern fonts, *roman*, *italic*, and bold (by Don Knuth) 6, 7, 8, 9, 10, 11, 12 point. (Available as cm)

Note that the cm fonts are intended for TEX and don't fare so well with troff. The spacing is not proportional by point size, and hence only one point size can be tuned to be nicely spaced. We have tuned the 10 point size, but the 8 point looks somewhat cramped.

Some of the punctuation is missing in some of the fonts. Knuth also uses a nonstandard notion of ASCII, and hence some glyphs are available only with special symbols such as \(12. Others cannot be accessed at all.

Knuth's fonts somewhat larger than normal, since he intends the output to be reduced before printing. Since troff has a limitation of 754 inches width on output, this is not practical. Hence, the original fonts have been relabelled with the point size they are closest to without reduction. Some fonts (6 point bold, 7 point roman, 8 point italic and bold, 9 point bold, and 11 point italic) which would have otherwise been missing were generated by shrinking the next larger point size of the same style. (This goes against the idea of metafont, but we use the tools we have.)

10 Point Roman

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234
56789 ! " # f% ' ( ) * - { } ^ ~ _ \ @ + / ? . > , < ', ,, Σ, Ξ, Υ, Φ, Π, Ψ, Δ, Θ, Λ, Ψ, Ω, ı, ȷ, ,, ,

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality since, as he elsewhere tells us, lost time is never found again and what we call time enough, always proves little enough Let us then up and be doing, and doing to the purpose so by diligence shall we do more with less perplexity.

10 Point Italic

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234
*56789 ! " # p ∂ ß ' ( ) : * - = [ ] { } ^ ~ _ \ ω @ ' ; + / ? . > , < ', ', Σ, -, -, Ξ, Υ, Φ, Π,*
η, θ, ι, Δ, Θ, Λ, Ψ, Ω, α, β, γ, ς ξ δ

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

10 Point Bold

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234
56789 ! " # ft % & ' ( ) : * - = [ ] { } ^ ~ _ \ ß @ ' ; + / ? . > , < ', ', Σ, -, -, Ξ, Υ, Φ, Π,
˘, ¯, ˜, Δ, Θ, Λ, Ψ, Ω, ı, ȷ, ˘, ˛, ˚ ˙ ˆ

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

6 Point Roman, Bold, and *Italic*
7 Point Roman, Bold, and *Italic*
8 Point Roman, Bold, and *Italic*
9 Point Roman, Bold, and *Italic*
10 Point Roman, Bold, and *Italic*
11 Point Roman, Bold, and *Italic*
12 Point Roman, Bold, and *Italic*

Countdown (22 point, upper case letters only.) From SAIL (Paul Martin)

ABCDE FGHIJ KLMNO PQRST UVWXYZ

COUNTDOWN HAS NO INTEGERS TO COUNT DOWN WITH BUT IT COMPENSATES BY BEING UGLY AND ILLEGIBLE

Cyrillic, 12 point only

ЖЦъЗ абде фгхи клмно прст увйя

ф тиме бе оф алл тхингс тхе мост преноус астинг тиме муст бе ас оор ихард сайс тхе греатест продигалитй сине ас хе елсехере телле ус лост тиме ис невер фоунд агаин анд хат е алл тиме еноугх алайс провес литтле еноугх ет ус тхен уп анд бе доинг анд доинг то тхе пурпосе со бй дилигене схалл е до море итх лесс перплекитй

W→Ж X→Ц Y→ъ Z→З a→а b→б d→д e→е f→ф g→г h→х i→и k→к l→л m→м n→н o→о p→п r→р s→с t→т u→у v→в y→й z→з

Delegate, roman, *italic*, and **bold**, 12 point only

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ↑ ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ↑ ; + / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ↑ ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Fix fixed width font, 6, 9, 10, 12, 14 point

6 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] _ | ' ; + / ? . >, <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

9 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

10 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

12 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity. ...

14 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Gacham, roman, bold, *italic*, 10 point only
The gacham font is almost indistinguishable from the fix font. In fact, it has been pointed out that our gacham roman and bold fonts really *are* fix. Sigh. They are included anyway for convenience.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789
*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789
! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Greek, 10 point only

This font provides an alternative to the Greek characters on the standard special font.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxy

ΑΒΧΔΕ ΦΓΗΙφ ΚΛΜΝΟ ΠΘΡΣΤ ΤΩΞΨΖ αδχδε φγηιδ ελμνο τθρστ υιωξψζ

Ιφ τιμε βε οφ αλλ τηινγσ τηε μοστ τρεχιουσ ωαστινγ τιμε μυστ βε, ασ Ποορ Ριχηαρδ σαψσ, τηε γρεατεστ προδιγαλιτψ; συνχε, ασ ηε ελσεωηερε τελλσ υσ, λοστ τιμε ισ νεϖερ φουνδ αγαιν; ανδ ωηατ ωε χαλλ τιμε ενουγη, αλωαψσ προϖεσ λιττλε ενουγη: Λετ υσ τηεν βε υπ ανδ βυσψ, ανδ δοινγ το τηε πυρποσε; σο βψ διλιγενχε σηαλλ ωε δο μορε ωιτη λεσσ περπλεξιτψ

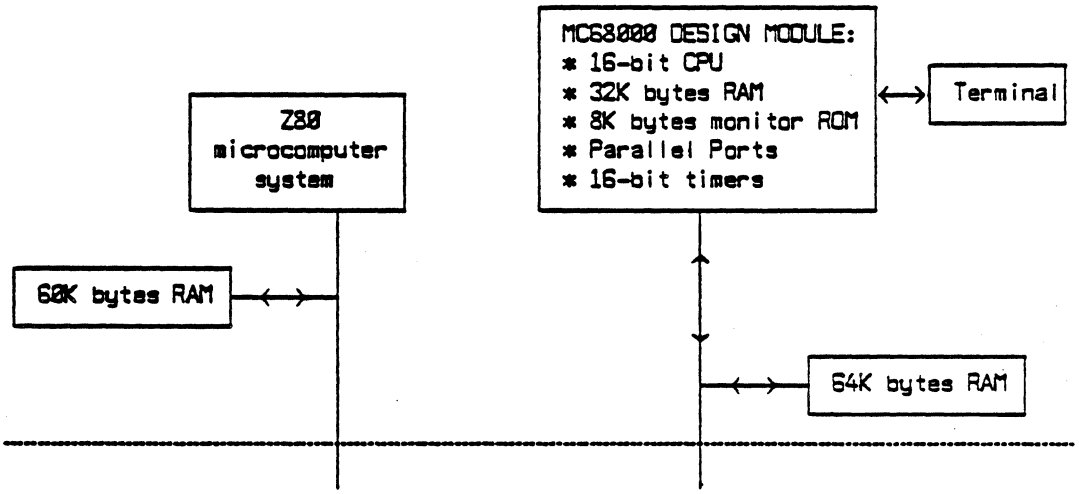
The h19 font includes a subset of the h19's graphic charactor set, plus a few logical extensions to allow forms and diagrams to be drawn. The charactors are the same as the h19's graphic interpratation set.

` a b c d e f s t u v m n h i k l

| ─ ┼ ┐ ┘ └ ┌ ┬ ┤ ┴ ├ ─ ┼ → ← ↓ ↑

The charactors are designed to overlap.

Example of usage for diagrams:



Hebrew, 16, 24, and 36 point only

16 point

יחתו שפ וחלכ יך בץ יחתו משרקפ ונמלכ עיהגפ עדסבא 01234 56789

! " # ף צ . ( ) : - [ ] { } ^ ~ _ \ | @ " : ץ ? . > , <

עהם שאה טב .שלטסבידערפ דנוחפורפ הפ עסבוהם טנעללעסחץ נא שב טנפ תודאהש עהט אואהץ שא ו .שחוכפ וחפ ף ו ולל וא שו וך תאצ .עלבארעדנו משטלא גניעכ פו ענאטנאדא לבאאן שטלץ וכ ף.

24 point

ם זיחתו טשרקפ נמלכ ע הג דסבא

&→צ .

דנ רפ ר ס הס טנ לל סח נא ש טנ ת דאהש הט
טש טלא גנ ב נאטנאודא הט שאה ט .שנ טס ד רפ
לבאדא רנ .

36 point (rather ragged)

זיחתו טשרקפ נמלכ ע הג דסבא

צ .

10 point Hershey

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789 !, $, %, &, ', (,), :, *, -, [,], ', :, /, ?, .

\(em → —, \- → -, \– → –, \(bu → •, \(sq → ▪, \(ru → _ \(14 → ¼, \(12 → ½, \(34 → ¾, \(fi → fi, \(fl → fl, \(ff → ff, \(Fi → ffi, \(Fl → ffl, \(de → °, \(dg → †, \(fm → ′, \(ct → ¢\(rg → ®\(co → ©

When you flex your fingers in a coffin, it can baffle a giraffe.

*ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789 !, $, %, &, ', (,), :, *, -, [,], ', :, /, ?, .*

\(em → —, \- → -, \– → –, \(bu → •, \(sq → ▪, \(ru → _ \(14 → ¼\(12 → ½\(34 → ¾\(fi → fi, \(fl → fl, \(ff → ff, \(Fi → ffi, \(Fl → ffl, \(de → °, \(dg → †, \(fm → ′, \(ct → ¢\(rg → ®\(co → ©

When you flex your fingers in a coffin, it can baffle a giraffe.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789 !, $, %, &, ', (,), :, *, -, [,], ', :, /, ?, .

\(em → —, \- → -, \– → –, \(bu → •, \(sq → ▪, \(ru → _, \(14 → ¼\(12 → ½\(34 → ¾\(fi → fi, \(fl → fl, \(ff → ff, \(Fi → ffi, \(Fl → ffl, \(de → °, \(dg → †, \(fm → ′, \(ct → ¢\(rg → ®\(co → ©

When you flex your fingers in a coffin, it can baffle a giraffe.
From special font: " # = { } ^ ~ _ \ | @ ` ´ + > <

Special characters: \(pl → +, \(mi → −, \(eq → =, \(** → *, \(sc → §, \(aa → ´, \(ga → `, \(ul → _, \(sl → /, \(*a → α, \(*b → β, \(*g → γ, \(*d → δ, \(*e → ε, \(*z → ζ, \(*y → η, \(*h → ϑ, \(*i → ι, \(*k → κ, \(*l → λ, \(*m → μ, \(*n → ν, \(*c → ξ, \(*o → ο, \(*p → π, \(*r → ρ, \(*s → σ, \(ts → ς, \(*t → τ, \(*u → υ, \(*f → φ, \(*x → χ, \(*q → ψ, \(*w → ω, \(*A → Α, \(*B → Β, \(*G → Γ, \(*D → Δ, \(*E → Ε, \(*Z → Ζ, \(*Y → Η, \(*H → Θ, \(*I → Ι, \(*K → Κ, \(*L → Λ, \(*M → Μ, \(*N → Ν, \(*C → Ξ, \(*O → Ο, \(*P → Π, \(*R → Ρ, \(*S → Σ, \(*T → Τ, \(*U → Υ, \(*F → Φ, \(*X → Χ, \(*Q → Ψ, \(*W → Ω, \(sr → √, \(rn → ¯, \(>= → ≥, \(<= → ≤, \(== → ≡, \(~= → ≅, \(ap → ~, \(!= → ≠, \(-> → →, \(<- → ←, \(ua → ↑, \(da → ↓, \(mu → ×, \(di → ÷, \(+- → ±, \(cu → ∪, \(ca → ∩, \(sb → ⊂, \(sp → ⊃, \(ib → ⊆, \(ip → ⊇, \(if → ∞, \(pd → ∂, \(gr → ∇, \(no → ¬, \(is → ∫, \(pt → ∝, \(eq → =, \(no → ¬, \(br → |, \(dd → ‡, \(rh → ☞\(lh → ☜, \(bs → \, \(or → |, \(ci → ○, \(lt → ⌈, \(lb → ⌊, \(rt → ⌉, \(rb → ⌋, \(lk → {, \(rk → }, \(bv → |, \(lf → ⌊, \(rf → ⌋, \(lc → ⌈, \(rc → ⌉

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

This is an *example* of a sample in various fonts.

Hershey font. This is the default font for vtroff. Roman, *Italic* and **Bold** in 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36 point. The following examples are 10 point.

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

6 point Roman, **Bold**, and *Italic*.
7 point Roman, **Bold**, and *Italic*.
8 point Roman, **Bold**, and *Italic*.
9 point Roman, **Bold**, and *Italic*.
10 point Roman, **Bold**, and *Italic*.
11 point Roman, **Bold**, and *Italic*.
12 point Roman, **Bold**, and *Italic*.
14 point Roman, **Bold**, and *Italic*.
16 point Roman, **Bold**, and *Italic*.
18 point Roman, **Bold**, and *Italic*.
20 point Roman, **Bold**, and *Italic*.
22 point Roman, **Bold**, and *Italic*.
24 point Roman, **Bold**, and *Italic*.
28 point Roman, **Bold**, and *Italic*.
36 point Roman, **Bold**, and *Italic*.

Meteor, roman, bold, *italic*, 8, 10, 12 point, no 12 point italic.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : ª - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : ª - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : ª - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Microgramma font, 10 point only

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

!"#$%&'():=-+[]{}^~_\|@';+/?.>,<

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Mona font, 24 point only

ABCDE FGHIJ KLMNO PQRST UVWXYZ
abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ ¢ & ' ( ) : - { } ^ ~ _ \ @ ; ? . > , <

Philadelphia is the most pecksniffian of American cities, and thus probably leads the world.

- H. L. Mencken

Nonie, roman, bold, *italic*, 8, 10, 12 point

8 point
ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij kimno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij kimno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij kimno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

10 point
ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij kimno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij kimno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij kimno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

12 point
ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Old English, 8, 14, and 18 point only. (This font is called "oldenglish" on line.)

8 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

" # ' :- { } ^ ~ _ \ @ ; .>.<

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality;since, as he elsewhere tells us, lost time is never found again;and what we call time enough, always proves little enough:Let us then up and be doing, and doing to the purpose;so by diligence shall we do more with less perplexity.

14 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

" # : { } ^ ~ _ \ @ ; .>.<

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality;since, as he elsewhere tells us, lost time is never found again;and what we call time enough, always proves little enough:Let us then up and be doing, and doing to the purpose;so by diligence shall we do more with less perplexity.

18 point

ABCDE FGHIJ KLMNO PQRST UVWXYZ
abcde fghij klmno pqrst uvwxyz 01234 56789

" # ' { } - { } ^ ~ - \ ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality since, as he elsewhere tells us, lost time is nver found again and what we call time enough, always proves little enough and I think I'm wasting time typing all this stuff

PIP FONT, 16 POINT ONLY, NO LOWER CASE

ABCDE FGHIJ KLMNO PQRST UVWXYZ 01234 56789

!"# '[]:- {}^~_\ @‘; ?.>,<

IT COULD PROBABLY BE SHOWN BY FACTS AND FIGURES THAT THERE IS NO DISTINCTLY NATIVE AMERICAN CRIMINAL CLASS EXCEPT CONGRESS.

– MARK TWAIN

Marhill font, 18 point only

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 0123

!"#$%&'():*-+[]{}^~_\ @ ;+/?.>,<

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Script, 18 point only. This font appears to be almost identical to the "Coronet" font from SAIL, except that the period and one other glyph of Coronet are missing a row, and Coronet is supposed to be 16 point. (They are both really the same size.)

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

"# : {}^~_\ @ ; .>,<

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

SHADOW, 16 POINT ONLY, NO LOWER CASE

ABCDE FGHIJ KLMNO PQRST UVWXYZ 01234 56789

!"# ' ; ⌧⌧{}~~_\ @',⌖ .>,<

THE SHADOW FONT IS AN EXCELLENT CHOICE FOR PROFOUND PREDICTIONS. IT HAS THE ADVANTAGE OF BEING ALMOST UNREADABLE.

⌖⌖ S H A D O W ⌖⌖

SIGN, 22 POINT ONLY

ABCDE FGHIJ KLMNO PQRST
UVWXYZ ➢➣ 0123456789

!"# ' :*-=. {}~~_@ ; / .>,<

THIS FONT WAS INVENTED BY A DRAFTSMAN WHO HAD LOST HIS FRENCH CURVE. ➢SO IT GOES ➣

LOWER CASE L IS ➢, LOWER CASE R IS ➣.

Stare hershey font. This font is identical to the hershey font except that the point sizes are one point smaller, and the width tables are those used for the real typesetter. Hence, this font is useful when previewing documents that are to be sent to a typesetter to make sure the spacing, paging, and so on is right. There are Roman, *Italic* and **Bold** in 8, 9, 10, 11, 12, 14, and 16 point. The following examples are 10 point.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + /? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + /? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + /? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

8 point Roman, **Bold**, and *Italic*.
9 point Roman, **Bold**, and *Italic*.
10 point Roman, **Bold**, and *Italic*.
11 point Roman, **Bold**, and *Italic*.
12 point Roman, **Bold**, and *Italic*.
14 point Roman, **Bold**, and *Italic*.
16 point Roman, **Bold**, and *Italic*.

Times fonts, roman, *italic*, and **bold**. 10 point only.
These fonts showed up in a directory labelled "timesroman" along with three other fonts which turned out to be nonie, meteor, and news gothic. They are probably not really times fonts, but seem to be pretty close. Notice the top of the "2" for a clear difference from a real Times Roman font.

It is our desire to have a real, digitized version of the times fonts from the phototypesetter. We eventually plan to do this. At that point, the times font will probably replace the hershey font as the default. Such a Times font is already available from Johns Hopkins University for a fee, but we couldn't redistribute it, so we plan do digitize them ourselves.

10 Point
ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789
! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <
', ', —, -, -, -, •, □, , ¼, ½, ¾, fi, fl, ff, ffi, ffl, °, †, ', ¢, ∞

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789
*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <*
', ', —, -, -, -, •, □, , ¼, ½, ¾, *fi*, *fl*, *ff*, *ffi*, *ffl*, °, †, ', ¢, ∞

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789
! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <
', ', —, -, -, -, •, □, , ¼, ½, ¾, fi, fl, ff, ffi, ffl, °, †, ', ¢, ∞

A Tour through the UNIX† C Compiler

D. M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

The Intermediate Language

Communication between the two phases of the compiler proper is carried out by means of a pair of intermediate files. These files are treated as having identical structure, although the second file contains only the code generated for strings. It is convenient to write strings out separately to reduce the need for multiple location counters in a later assembly phase.

The intermediate language is not machine-independent; its structure in a number of ways reflects the fact that C was originally a one-pass compiler chopped in two to reduce the maximum memory requirement. In fact, only the latest version of the compiler has a complete intermediate language at all. Until recently, the first phase of the compiler generated assembly code for those constructions it could deal with, and passed expression parse trees, in absolute binary form, to the second phase for code generation. Now, at least, all inter-phase information is passed in a describable form, and there are no absolute pointers involved, so the coupling between the phases is not so strong.

The areas in which the machine (and system) dependencies are most noticeable are

1. Storage allocation for automatic variables and arguments has already been performed, and nodes for such variables refer to them by offset from a display pointer. Type conversion (for example, from integer to pointer) has already occurred using the assumption of byte addressing and 2-byte words.
2. Data representations suitable to the PDP-11 are assumed; in particular, floating point constants are passed as four words in the machine representation.

As it happens, each intermediate file is represented as a sequence of binary numbers without any explicit demarcations. It consists of a sequence of conceptual lines, each headed by an operator, and possibly containing various operands. The operators are small numbers; to assist in recognizing failure in synchronization, the high-order byte of each operator word is always the octal number 376. Operands are either 16-bit binary numbers or strings of characters representing names. Each name is terminated by a null character. There is no alignment requirement for numerical operands and so there is no padding after a name string.

The binary representation was chosen to avoid the necessity of converting to and from character form and to minimize the size of the files. It would be very easy to make each operator-operand 'line' in the file be a genuine, printable line, with the numbers in octal or decimal; this in fact was the representation originally used.

The operators fall naturally into two classes: those which represent part of an expression, and all others. Expressions are transmitted in a reverse-Polish notation; as they are being read, a tree is built which is isomorphic to the tree constructed in the first phase. Expressions are passed as a whole, with no non-expression operators intervening. The reader maintains a stack; each leaf of the expression tree (name, constant) is pushed on the stack; each unary operator replaces the top of the stack by a node whose operand is the old top-of-stack; each binary

operator replaces the top pair on the stack with a single entry. When the expression is complete there is exactly one item on the stack. Following each expression is a special operator which passes the unique previous expression to the 'optimizer' described below and then to the code generator.

Here is the list of operators not themselves part of expressions.

EOF

marks the end of an input file.

BDATA *flag data ...*

specifies a sequence of bytes to be assembled as static data. It is followed by pairs of words; the first member of the pair is non-zero to indicate that the data continue; a zero flag is not followed by data and terminates the operator. The data bytes occupy the low-order part of a word.

WDATA *flag data ...*

specifies a sequence of words to be assembled as static data; it is identical to the BDATA operator except that entire words, not just bytes, are passed.

PROG

means that subsequent information is to be compiled as program text.

DATA

means that subsequent information is to be compiled as static data.

BSS

means that subsequent information is to be compiled as unitialized static data.

SYMDEF *name*

means that the symbol *name* is an external name defined in the current program. It is produced for each external data or function definition.

CSPACE *name size*

indicates that the name refers to a data area whose size is the specified number of bytes. It is produced for external data definitions without explicit initialization.

SSPACE *size*

indicates that *size* bytes should be set aside for data storage. It is used to pad out short initializations of external data and to reserve space for static (internal) data. It will be preceded by an appropriate label.

EVEN

is produced after each external data definition whose size is not an integral number of words. It is not produced after strings except when they initialize a character array.

NLABEL *name*

is produced just before a BDATA or WDATA initializing external data, and serves as a label for the data.

RLABEL *name*

is produced just before each function definition, and labels its entry point.

SNAME *name number*

is produced at the start of each function for each static variable or label declared therein. Subsequent uses of the variable will be in terms of the given number. The code generator uses this only to produce a debugging symbol table.

ANAME *name number*

Likewise, each automatic variable's name and stack offset is specified by this operator. Arguments count as automatics.

RNAME *name number*

Each register variable is similarly named, with its register number.

SAVE *number*

produces a register-save sequence at the start of each function, just after its label (RLABEL).

SETREG *number*

is used to indicate the number of registers used for register variables. It actually gives the register number of the lowest free register; it is redundant because the RNAME operators could be counted instead.

PROFIL

is produced before the save sequence for functions when the profile option is turned on. It produces code to count the number of times the function is called.

SWIT *deflab line label value ...*

is produced for switches. When control flows into it, the value being switched on is in the register forced by RFORCE (below). The switch statement occurred on the indicated line of the source, and the label number of the default location is *deflab*. Then the operator is followed by a sequence of label-number and value pairs; the list is terminated by a 0 label.

LABEL *number*

generates an internal label. It is referred to elsewhere using the given number.

BRANCH *number*

indicates an unconditional transfer to the internal label number given.

RETRN

produces the return sequence for a function. It occurs only once, at the end of each function.

EXPR *line*

causes the expression just preceding to be compiled. The argument is the line number in the source where the expression occurred.

NAME *class type name*

NAME *class type number*

indicates a name occurring in an expression. The first form is used when the name is external; the second when the name is automatic, static, or a register. Then the number indicates the stack offset, the label number, or the register number as appropriate. Class and type encoding is described elsewhere.

CON *type value*

transmits an integer constant. This and the next two operators occur as part of expressions.

FCON *type 4-word-value*

transmits a floating constant as four words in PDP-11 notation.

SFCON *type value*

transmits a floating-point constant whose value is correctly represented by its high-order word in PDP-11 notation.

NULL

indicates a null argument list of a function call in an expression; call is a binary operator whose second operand is the argument list.

CBRANCH *label cond*

produces a conditional branch. It is an expression operator, and will be followed by an EXPR. The branch to the label number takes place if the expression's truth value is the same as that of *cond*. That is, if *cond*=*1* and the expression evaluates to true, the branch is taken.

binary-operator *type*

There are binary operators corresponding to each such source-language operator; the type of the result of each is passed as well. Some perhaps-unexpected ones are: COMMA, which is a right-associative operator designed to simplify right-to-left evaluation of function arguments; prefix and postfix ++ and --, whose second operand is the increment amount, as a CON; QUEST and COLON, to express the conditional expression as 'a?(b:c)'; and a sequence of special operators for expressing relations between pointers, in case pointer comparison is different from integer comparison (e.g. unsigned).

unary-operator *type*

There are also numerous unary operators. These include ITOF, FTOI, FTOL, LTOF, ITOL, LTOI which convert among floating, long, and integer; JUMP which branches indirectly through a label expression; INIT, which compiles the value of a constant expression used as an initializer; RFORCE, which is used before a return sequence or a switch to place a value in an agreed-upon register.

Expression Optimization

Each expression tree, as it is read in, is subjected to a fairly comprehensive analysis. This is performed by the *optim* routine and a number of subroutines; the major things done are

1. Modifications and simplifications of the tree so its value may be computed more efficiently and conveniently by the code generator.
2. Marking each interior node with an estimate of the number of registers required to evaluate it. This register count is needed to guide the code generation algorithm.

One thing that is definitely not done is discovery or exploitation of common subexpressions, nor is this done anywhere in the compiler.

The basic organization is simple: a depth-first scan of the tree. *Optim* does nothing for leaf nodes (except for automatics; see below), and calls *unoptim* to handle unary operators. For binary operators, it calls itself to process the operands, then treats each operator separately. One important case is commutative and associative operators, which are handled by *acommute*.

Here is a brief catalog of the transformations carried out by by *optim* itself. It is not intended to be complete. Some of the transformations are machine-dependent, although they may well be useful on machines other than the PDP-11.

1. As indicated in the discussion of *unoptim* below, the optimizer can create a node type corresponding to the location addressed by a register plus a constant offset. Since this is precisely the implementation of automatic variables and arguments, where the register is fixed by convention, such variables are changed to the new form to simplify later processing.
2. Associative and commutative operators are processed by the special routine *acommute*.
3. After processing by *acommute*, the bitwise & operator is turned into a new *andn* operator; 'a & b' becomes 'a *andn* ~b'. This is done because the PDP-11 provides no *and* operator, but only *andn*. A similar transformation takes place for '=&'.
4. Relationals are turned around so the more complicated expression is on the left. (So that '2 > f(x)' becomes 'f(x) < 2'). This improves code generation since the algorithm prefers to have the right operand require fewer registers than the left.
5. An expression minus a constant is turned into the expression plus the negative constant, and the *acommute* routine is called to take advantage of the properties of addition.
6. Operators with constant operands are evaluated.
7. Right shifts (unless by 1) are turned into left shifts with a negated right operand, since the PDP-11 lacks a general right-shift operator.
8. A number of special cases are simplified, such as division or multiplication by 1, and shifts by 0.

The *unoptim* routine performs the same sort of processing for unary operators.

1. '*&x' and '&*x' are simplified to 'x'.
2. If *r* is a register and *c* is a constant or the address of a static or external variable, the expressions '*(r+c)' and '*r' are turned into a special kind of name node which expresses the name itself and the offset. This simplifies subsequent processing because such constructions can appear as the the address of a PDP-11 instruction.
3. When the unary '&' operator is applied to a name node of the special kind just discussed, it is reworked to make the addition explicit again; this is done because the PDP-11 has no 'load address' instruction.
4. Constructions like '*r++' and '*--r' where *r* is a register are discovered and marked as being implementable using the PDP-11 auto-increment and -decrement modes.
5. If '!' is applied to a relational, the '!' is discarded and the sense of the relational is reversed.
6. Special cases involving reflexive use of negation and complementation are discovered.

7. Operations applying to constants are evaluated.

The *acommute* routine, called for associative and commutative operators, discovers clusters of the same operator at the top levels of the current tree, and arranges them in a list: for 'a+((b+c)+(d+f))' the list would be'a,b,c,d,e,f'. After each subtree is optimized, the list is sorted in decreasing difficulty of computation; as mentioned above, the code generation algorithm works best when left operands are the difficult ones. The 'degree of difficulty' computed is actually finer than the mere number of registers required; a constant is considered simpler than the address of a static or external, which is simpler than reference to a variable. This makes it easy to fold all the constants together, and also to merge together the sum of a constant and the address of a static or external (since in such nodes there is space for an 'offset' value). There are also special cases, like multiplication by 1 and addition of 0.

A special routine is invoked to handle sums of products. *Distrib* is based on the fact that it is better to compute 'c1*c2*x + c1*y' as 'c1*(c2*x + y)' and makes the divisibility tests required to assure the correctness of the transformation. This transformation is rarely possible with code directly written by the user, but it invariably occurs as a result of the implementation of multi-dimensional arrays.

Finally, *acommute* reconstructs a tree from the list of expressions which result.

Code Generation

The grand plan for code-generation is independent of any particular machine; it depends largely on a set of tables. But this fact does not necessarily make it very easy to modify the compiler to produce code for other machines, both because there is a good deal of machine-dependent structure in the tables, and because in any event such tables are non-trivial to prepare.

The arguments to the basic code generation routine *rcexpr* are a pointer to a tree representing an expression, the name of a code-generation table, and the number of a register in which the value of the expression should be placed. *Rcexpr* returns the number of the register in which the value actually ended up; its caller may need to produce a *mov* instruction if the value really needs to be in the given register. There are four code generation tables.

Regtab is the basic one, which actually does the job described above: namely, compile code which places the value represented by the expression tree in a register.

Cctab is used when the value of the expression is not actually needed, but instead the value of the condition codes resulting from evaluation of the expression. This table is used, for example, to evaluate the expression after *if*. It is clearly silly to calculate the value (0 or 1) of the expression 'a==b' in the context 'if (a==b) ... '

The *sptab* table is used when the value of an expression is to be pushed on the stack, for example when it is an actual argument. For example in the function call 'f(a)' it is a bad idea to load *a* into a register which is then pushed on the stack, when there is a single instruction which does the job.

The *efftab* table is used when an expression is to be evaluated for its side effects, not its value. This occurs mostly for expressions which are statements, which have no value. Thus the code for the statement 'a = b' need produce only the approoriate *mov* instruction, and need not leave the value of *b* in a register, while in the expression 'a + (b = c)' the value of 'b = c' will appear in a register.

All of the tables besides *regtab* are rather small, and handle only a relatively few special cases. If one of these subsidiary tables does not contain an entry applicable to the given expression tree, *rcexpr* uses *regtab* to put the value of the expression into a register and then fixes things up; nothing need be done when the table was *efftab*, but a *tst* instruction is produced when the table called for was *cctab*, and a *mov* instruction, pushing the register on the stack, when the table was *sptab*.

The *rcexpr* routine itself picks off some special cases, then calls *cexpr* to do the real work. *Cexpr* tries to find an entry applicable to the given tree in the given table, and returns −1 if no such entry is found, letting *rcexpr* try again with a different table. A successful match yields a string containing both literal characters which are written out and pseudo-operations, or macros, which are expanded. Before studying the contents of these strings we will consider how table entries are matched against trees.

Recall that most non-leaf nodes in an expression tree contain the name of the operator, the type of the value represented, and pointers to the subtrees (operands). They also contain an estimate of the number of registers required to evaluate the expression, placed there by the expression-optimizer routines. The register counts are used to guide the code generation process, which is based on the Sethi-Ullman algorithm.

The main code generation tables consist of entries each containing an operator number and a pointer to a subtable for the corresponding operator. A subtable consists of a sequence of entries, each with a key describing certain properties of the operands of the operator involved; associated with the key is a code string. Once the subtable corresponding to the operator is found, the subtable is searched linearly until a key is found such that the properties demanded by the key are compatible with the operands of the tree node. A successful match returns the code string; an unsuccessful search, either for the operator in the main table or a compatble key in the subtable, returns a failure indication.

The tables are all contained in a file which must be processed to obtain an assembly language program. Thus they are written in a special-purpose language. To provided definiteness to the following discussion, here is an example of a subtable entry.

```
%n,aw
        F
        add     A2,R
```

The '%' indicates the key; the information following (up to a blank line) specifies the code string. Very briefly, this entry is in the subtable for '+' of *regtab;* the key specifies that the left operand is any integer, character, or pointer expression, and the right operand is any word quantity which is directly addressible (e.g. a variable or constant). The code string calls for the generation of the code to compile the left (first) operand into the current register ('F') and then to produce an 'add' instruction which adds the second operand ('A2') to the register ('R'). All of the notation will be explained below.

Only three features of the operands are used in deciding whether a match has occurred. They are:

1. Is the type of the operand compatible with that demanded?
2. Is the 'degree of difficulty' (in a sense described below) compatible?
3. The table may demand that the operand have a '*' (indirection operator) as its highest operator.

As suggested above, the key for a subtable entry is indicated by a '%,' and a comma-separated pair of specifications for the operands. (The second specification is ignored for unary operators). A specification indicates a type requirement by including one of the following letters. If no type letter is present, any integer, character, or pointer operand will satisfy the requirement (not float, double, or long).

b A byte (character) operand is required.

w A word (integer or pointer) operand is required.

f A float or double operand is required.

d A double operand is required.

l A long (32-bit integer) operand is required.

Before discussing the 'degree of difficulty' specification, the algorithm has to be explained more completely. *Rcexpr* (and *cexpr)* are called with a register number in which to place their result. Registers 0, 1, ... are used during evaluation of expressions; the maximum register which can be used in this way depends on the number of register variables, but in any event only registers 0 through 4 are available since r5 is used as a stack frame header and r6 (sp) and r7 (pc) have special hardware properties. The code generation routines assume that when called with register *n* as argument, they may use *n+1*, ... (up to the first register variable) as temporaries. Consider the expression 'X+Y', where both X and Y are expressions. As a first approximation, there are three ways of compiling code to put this expression in register *n*.

1. If Y is an addressible cell, (recursively) put X into register *n* and add Y to it.
2. If Y is an expression that can be calculated in *k* registers, where *k* smaller than the number of registers available, compile X into register *n*, Y into register *n+1*, and add register *n+1* to *n*.
3. Otherwise, compile Y into register *n*, save the result in a temporary (actually, on the stack) compile X into register *n*, then add in the temporary.

The distinction between cases 2 and 3 therefore depends on whether the right operand can be compiled in fewer than *k* registers, where *k* is the number of free registers left after registers 0 through *n* are taken: 0 through *n−1* are presumed to contain already computed temporary results; *n* will, in case 2, contain the value of the left operand while the right is being evaluated.

These considerations should make clear the specification codes for the degree of difficulty, bearing in mind that a number of special cases are also present:

z is satisfied when the operand is zero, so that special code can be produced for expressions like 'x = 0'.

1 is satisfied when the operand is the constant 1, to optimize cases like left and right shift by 1, which can be done efficiently on the PDP-11.

c is satisfied when the operand is a positive (16-bit) constant; this takes care of some special cases in long arithmetic.

a is satisfied when the operand is addressible; this occurs not only for variables and constants, but also for some more complicated constructions, such as indirection through a simple variable, '*p++' where *p* is a register variable (because of the PDP-11's auto-increment address mode), and '*(p+c)' where *p* is a register and *c* is a constant. Precisely, the requirement is that the operand refers to a cell whose address can be written as a source or destination of a PDP-11 instruction.

e is satisfied by an operand whose value can be generated in a register using no more than *k* registers, where *k* is the number of registers left (not counting the current register). The 'e' stands for 'easy.'

n is satisfied by any operand. The 'n' stands for 'anything.'

These degrees of difficulty are considered to lie in a linear ordering and any operand which satisfies an earlier-mentioned requirement will satisfy a later one. Since the subtables are searched linearly, if a '1' specification is included, almost certainly a 'z' must be written first to prevent expressions containing the constant 0 to be compiled as if the 0 were 1.

Finally, a key specification may contain a '*' which requires the operand to have an indirection as its leading operator. Examples below should clarify the utility of this specification.

Now let us consider the contents of the code string associated with each subtable entry. Conventionally, lower-case letters in this string represent literal information which is copied directly to the output. Upper-case letters generally introduce specific macro-operations, some of which may be followed by modifying information. The code strings in the tables are written with tabs and new-lines used freely to suggest instructions which will be generated; the table-

compiling program compresses tabs (using the 0200 bit of the next character) and throws away some of the new-lines. For example the macro 'F' is ordinarily written on a line by itself; but since its expansion will end with a new-line, the new-line after 'F' itself is dispensable. This is all to reduce the size of the stored tables.

The first set of macro-operations is concerned with compiling subtrees. Recall that this is done by the *cexpr* routine. In the following discussion the 'current register' is generally the argument register to *cexpr;* that is, the place where the result is desired. The 'next register' is numbered one higher than the current register. (This explanation isn't fully true because of complications, described below, involving operations which require even-odd register pairs.)

F causes a recursive call to the *rcexpr* routine to compile code which places the value of the first (left) operand of the operator in the current register.

F1 generates code which places the value of the first operand in the next register. It is incorrectly used if there might be no next register; that is, if the degree of difficulty of the first operand is not 'easy;' if not, another register might not be available.

FS generates code which pushes the value of the first operand on the stack, by calling *rcexpr* specifying *sptab* as the table.

Analogously,

S, S1, SScompile the second (right) operand into the current register, the next register, or onto the stack.

To deal with registers, there are

R which expands into the name of the current register.

R1 which expands into the name of the next register.

R+ which expands into the the name of the current register plus 1. It was suggested above that this is the same as the next register, except for complications; here is one of them. Long integer variables have 32 bits and require 2 registers; in such cases the next register is the current register plus 2. The code would like to talk about both halves of the long quantity, so R refers to the register with the high-order part and R+ to the low-order part.

R− This is another complication, involving division and mod. These operations involve a pair of registers of which the odd-numbered contains the left operand. *Cexpr* arranges that the current register is odd; the R− notation allows the code to refer to the next lower, even-numbered register.

To refer to addressible quantities, there are the notations:

A1 causes generation of the address specified by the first operand. For this to be legal, the operand must be addressible; its key must contain an 'a' or a more restrictive specification.

A2 correspondingly generates the address of the second operand providing it has one.

We now have enough mechanism to show a complete, if suboptimal, table for the + operator on word or byte operands.

```
%n,z
        F

%n,1
        F
        inc     R

%n,aw
        F
        add     A2,R

%n,e
        F
        S1
        add     R1,R

%n,n
        SS
        F
        add     (sp)+,R
```

The first two sequences handle some special cases. Actually it turns out that handling a right operand of 0 is unnecessary since the expression-optimizer throws out adds of 0. Adding 1 by using the 'increment' instruction is done next, and then the case where the right operand is addressible. It must be a word quantity, since the PDP-11 lacks an 'add byte' instruction. Finally the cases where the right operand either can, or cannot, be done in the available registers are treated.

The next macro-instructions are conveniently introduced by noticing that the above table is suitable for subtraction as well as addition, since no use is made of the commutativity of addition. All that is needed is substitution of 'sub' for 'add' and 'dec' for 'inc.' Considerable saving of space is achieved by factoring out several similar operations.

I is replaced by a string from another table indexed by the operator in the node being expanded. This secondary table actually contains two strings per operator.

I' is replaced by the second string in the side table entry for the current operator.

Thus, given that the entries for '+' and '−' in the side table (which is called *instab)* are 'add' and 'inc,' 'sub' and 'dec' respectively, the middle of of the above addition table can be written

```
%n,1
        F
        I'      R

%n,aw
        F
        I       A2,R
```

and it will be suitable for subtraction, and several other operators, as well.

Next, there is the question of character and floating-point operations.

B1 generates the letter 'b' if the first operand is a character, 'f' if it is float or double, and nothing otherwise. It is used in a context like 'movB1' which generates a 'mov', 'movb', or 'movf' instruction according to the type of the operand.

B2 is just like B1 but applies to the second operand.

BE generates 'b' if either operand is a character and null otherwise.

BF generates 'f' if the type of the operator node itself is float or double, otherwise null.

For example, there is an entry in *effiab* for the '=' operator

```
%a,aw
%ab,a
        IBE     A2,A1
```

Note first that two key specifications can be applied to the same code string. Next, observe that when a word is assigned to a byte or to a word, or a word is assigned to a byte, a single instruction, a *mov* or *movb* as appropriate, does the job. However, when a byte is assigned to a word, it must pass through a register to implement the sign-extension rules:

```
%a,n
        S
        IB1     R,A1
```

Next, there is the question of handling indirection properly. Consider the expression 'X + *Y', where X and Y are expressions, Assuming that Y is more complicated than just a variable, but on the other hand qualifies as 'easy' in the context, the expression would be compiled by placing the value of X in a register, that of *Y in the next register, and adding the registers. It is easy to see that a better job can be done by compiling X, then Y (into the next register), and producing the instruction symbolized by 'add (R1),R'. This scheme avoids generating the instruction 'mov (R1),R1' required actually to place the value of *Y in a register. A related situation occurs with the expression 'X + *(p+6)', which exemplifies a construction frequent in structure and array references. The addition table shown above would produce

```
[put X in register R]
mov     p,R1
add     $6,R1
mov     (R1),R1
add     R1,R
```

when the best code is

```
[put X in R]
mov     p,R1
add     6(R1),R
```

As we said above, a key specification for a code table entry may require an operand to have an indirection as its highest operator. To make use of the requirement, the following macros are provided.

F* the first operand must have the form *X. If in particular it has the form *(Y + c), for some constant *c*, then code is produced which places the value of Y in the current register. Otherwise, code is produced which loads X into the current register.

F1* resembles F* except that the next register is loaded.

S* resembles F* except that the second operand is loaded.

S1* resembles S* except that the next register is loaded.

FS* The first operand must have the form '*X'. Push the value of X on the stack.

SS* resembles FS* except that it applies to the second operand.

To capture the constant that may have been skipped over in the above macros, there are

#1 The first operand must have the form *X; if in particular it has the form *(Y + c) for *c* a constant, then the constant is written out, otherwise a null string.

#2 is the same as #1 except that the second operand is used.

Now we can improve the addition table above. Just before the '%n,e' entry, put

```
%n,ew*
        F
        S1*
        add     #2(R1),R
```

and just before the '%n,n' put

```
%n,nw*
        SS*
        F
        add     *(sp)+,R
```

When using the stacking macros there is no place to use the constant as an index word, so that particular special case doesn't occur.

The constant mentioned above can actually be more general than a number. Any quantity acceptable to the assembler as an expression will do, in particular the address of a static cell, perhaps with a numeric offset. If *x* is an external character array, the expression 'x[i+5] = 0' will generate the code

```
mov     i,r0
clrb    x+5(r0)
```

via the table entry (in the '=' part of *effiab)*

```
%e*,z
        F
        I'B1    #1(R)
```

Some machine operations place restrictions on the registers used. The divide instruction, used to implement the divide and mod operations, requires the dividend to be placed in the odd member of an even-odd pair; other peculiarities of multiplication make it simplest to put the multiplicand in an odd-numbered register. There is no theory which optimally accounts for this kind of requirement. *Cexpr* handles it by checking for a multiply, divide, or mod operation; in these cases, its argument register number is incremented by one or two so that it is odd, and if the operation was divide or mod, so that it is a member of a free even-odd pair. The routine which determines the number of registers required estimates, conservatively, that at least two registers are required for a multiplication and three for the other peculiar operators. After the expression is compiled, the register where the result actually ended up is returned. (Divide and mod are actually the same operation except for the location of the result).

These operations are the ones which cause results to end up in unexpected places, and this possibility adds a further level of complexity. The simplest way of handling the problem is always to move the result to the place where the caller expected it, but this will produce unnecessary register moves in many simple cases; 'a = b*c' would generate

```
mov     b,r1
mul     c,r1
mov     r1,r0
mov     r0,a
```

The next thought is used the passed-back information as to where the result landed to change the notion of the current register. While compiling the '=' operation above, which comes from a table entry like

```
%a,e
        S
        mov     R,A1
```

it is sufficient to redefine the meaning of 'R' after processing the 'S' which does the multiply. This technique is in fact used; the tables are written in such a way that correct code is produced. The trouble is that the technique cannot be used in general, because it invalidates the count of the number of registers required for an expression. Consider just 'a*b + X' where X is some expression. The algorithm assumes that the value of a*b, once computed, requires just one register. If there are three registers available, and X requires two registers to compute, then this expression will match a key specifying '%n,e'. If a*b is computed and left in register 1, then there are, contrary to expectations, no longer two registers available to compute X, but only one, and bad code will be produced. To guard against this possibility, *cexpr* checks the result returned by recursive calls which implement F, S and their relatives. If the result is not in the expected register, then the number of registers required by the other operand is checked; if it can be done using those registers which remain even after making unavailable the unexpectedly-occupied register, then the notions of the 'next register' and possibly the 'current register' are redefined. Otherwise a register-copy instruction is produced. A register-copy is also always produced when the current operator is one of those which have odd-even requirements.

Finally, there are a few loose-end macro operations and facts about the tables. The operators:

V is used for long operations. It is written with an address like a machine instruction; it expands into 'adc' (add carry) if the operation is an additive operator, 'sbc' (subtract carry) if the operation is a subtractive operator, and disappears, along with the rest of the line, otherwise. Its purpose is to allow common treatment of logical operations, which have no carries, and additive and subtractive operations, which generate carries.

T generates a 'tst' instruction if the first operand of the tree does not set the condition codes correctly. It is used with divide and mod operations, which require a sign-extended 32-bit operand. The code table for the operations contains an 'sxt' (sign-extend) instruction to generate the high-order part of the dividend.

H is analogous to the 'F' and 'S' macros, except that it calls for the generation of code for the current tree (not one of its operands) using *regtab*. It is used in *cctab* for all the operators which, when executed normally, set the condition codes properly according to the result. It prevents a 'tst' instruction from being generated for constructions like 'if (a+b) ...' since after calculation of the value of 'a+b' a conditional branch can be written immediately.

All of the discussion above is in terms of operators with operands. Leaves of the expression tree (variables and constants), however, are peculiar in that they have no operands. In order to regularize the matching process, *cexpr* examines its operand to determine if it is a leaf; if so, it creates a special 'load' operator whose operand is the leaf, and substitutes it for the argument tree; this allows the table entry for the created operator to use the 'A1' notation to load the leaf into a register.

Purely to save space in the tables, pieces of subtables can be labelled and referred to later. It turns out, for example, that rather large portions of the the *effiab* table for the '=' and '=+' operators are identical. Thus '=' has an entry

```
%[move3:]
%a,aw
%ab,a
        IBE     A2,A1
```

while part of the '=+' table is

```
%aw,aw
%       [move3]
```

Labels are written as '%[... :]', before the key specifications; references are written with '% [...]' after the key. Peculiarities in the implementation make it necessary that labels appear before references to them.

The example illustrates the utility of allowing separate keys to point to the same code string. The assignment code works properly if either the right operand is a word, or the left operand is a byte; but since there is no 'add byte' instruction the addition code has to be restricted to word operands.

Delaying and reordering

Intertwined with the code generation routines are two other, interrelated processes. The first, implemented by a routine called *delay*, is based on the observation that naive code generation for the expression 'a = b++' would produce

```
mov     b,r0
inc     b
mov     r0,a
```

The point is that the table for postfix ++ has to preserve the value of *b* before incrementing it; the general way to do this is to preserve its value in a register. A cleverer scheme would generate

```
mov     b,a
inc     b
```

Delay is called for each expression input to *rcexpr*, and it searches for postfix ++ and −− operators. If one is found applied to a variable, the tree is patched to bypass the operator and compiled as it stands; then the increment or decrement itself is done. The effect is as if 'a = b; b++' had been written. In this example, of course, the user himself could have done the same job, but more complicated examples are easily constructed, for example 'switch (x++)'. An essential restriction is that the condition codes not be required. It would be incorrect to compile 'if (a++) ...' as

```
tst     a
inc     a
beq     ...
```

because the 'inc' destroys the required setting of the condition codes.

Reordering is a similar sort of optimization. Many cases which it detects are useful mainly with register variables. If *r* is a register variable, the expression 'r = x+y' is best compiled as

```
mov     x,r
add     y,r
```

but the codes tables would produce

```
mov     x,r0
add     y,r0
mov     r0,r
```

which is in fact preferred if *r* is not a register. (If *r* is not a register, the two sequences are the same size, but the second is slightly faster.) The scheme is to compile the expression as if it had been written 'r = x; r =+ y'. The *reorder* routine is called with a pointer to each tree that *rcexpr* is about to compile; if it has the right characteristics, the 'r = x' tree is constructed and passed recursively to *rcexpr*; then the original tree is modified to read 'r =+ y' and the calling instance of *rcexpr* compiles that instead. Of course the whole business is itself recursive so that

more extended forms of the same phenomenon are handled, like 'r = x + y | z'.

Care does have to be taken to avoid 'optimizing' an expression like 'r = x + r' into 'r = x; r =+ r'. It is required that the right operand of the expression on the right of the '=' be a ', distinct from the register variable.

The second case that *reorder* handles is expressions of the form 'r = X' used as a subexpression. Again, the code out of the tables for 'x = r = y' would be

```
mov   y,r0
mov   r0,r
mov   r0,x
```

whereas if *r* were a register it would be better to produce

```
mov   y,r
mov   r,x
```

When *reorder* discovers that a register variable is being assigned to in a subexpression, it calls *rcexpr* recursively to compile the subexpression, then fiddles the tree passed to it so that the register variable itself appears as the operand instead of the whole subexpression. Here care has to be taken to avoid an infinite regress, with *rcexpr* and *reorder* calling each other forever to handle assignments to registers.

A third set of cases treated by *reorder* comes up when any name, not necessarily a register, occurs as a left operand of an assignment operator other than '=' or as an operand of prefix '++' or '--'. Unless condition-code tests are involved, when a subexpression like '(a =+ b)' is seen, the assignment is performed and the argument tree modified so that *a* is its operand; effectively 'x + (y =+ z)' is compiled as 'y =+ z; x + y'. Similarly, prefix increment and decrement are pulled out and performed first, then the remainder of the expression.

Throughout code generation, the expression optimizer is called whenever *delay* or *reorder* change the expression tree. This allows some special cases to be found that otherwise would not be seen.

NAME

make — maintain, update, and regenerate groups of programs

SYNOPSIS

make [—f makefile] [—p] [—i] [—k] [—s] [—r] [—n] [—b] [—e] [—m] [—t] [—d] [—q] [names]

DESCRIPTION

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. A file name of — denotes the standard input. The contents of *makefile* override the built-in rules if they are present.

—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, 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.

—m Print a memory map showing text, data, and stack. This option is a no-operation on systems without the *getu* system call.

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

—d Debug mode. Print out detailed information on files and times examined.

—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**, **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 it depends on files that are newer than the target. All prerequisite files of a target are added recursively to the list of targets. Missing files are deemed to be 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 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 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. The difference between the old version of *make* and this version is that this version requires all dependency lines to have a (possibly null or implicit) command associated with them. The previous version of *make* assumed if no command was specified explicitly that the command was null.

Interrupt and quit cause the target to be deleted unless the target is a dependency 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.

The **MAKEFLAGS** environment variable is processed by *make* as containing any legal input option (except —**f**, —**p**, and —**d**) 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.

Macros

Entries of the form *string1* = *string2* are macro definitions. *String2* is defined as all characters up to a comment character or an unescaped newline. 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 **$?**. The reasons for this are debatable.

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~ .sh .sh~ .c.o .c~.o .c~.c .s.o .s~.o .y.o .y~.o .l.o .l~.o
.y.c .y~.c .l.c .c.a .c~.a .s~.a .h~.h

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

The only peculiarity in this output is the **(null)** string which *printf*(3S) prints when handed a null string.

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 .y .l .s

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 parenthesis, it is assumed to be an archive library, the string within parenthesis 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 rv $@ $*.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 rv 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

SEE ALSO

sh(1).
Make—A Program for Maintaining Computer Programs by S. I. Feldman.
An Augmented Version of Make by E. G. Bradford.

BUGS

Some commands return non-zero status inappropriately; use **—i** to overcome the difficulty. Commands that are directly executed by the shell, notably *cd*(1), are ineffectual across new-lines in *make*. The syntax **(lib(file1.o file2.o file3.o)** is illegal. You cannot build **lib(file.o)** from **file.o**. The macro **$(a:.o=.c~)** doesn't work.

Chapter 2

A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS (make)

PAGE

Chapter 2

A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS (make)

GENERAL

In a programming project, a common practice is to divide large programs into smaller pieces that are more manageable. The pieces may require several different treatments such as being processed by a macro processor or sophisticated program generators (e.g., **Yacc** or **Lex**). The project continues to become more complex as the output of these generators are compiled with special options and with certain definitions and declarations. A sequence of code transformations develops which is difficult to remember. The resulting code may need further transformation by loading the code with certain libraries under control of special options. Related maintenance activities also complicate the process further by running test scripts and installing validated modules. Another activity that complicates program development is a long editing session. A programmer may lose track of the files changed and the object modules still valid especially when a change to a declaration can make a dozen other files obsolete. The programmer must also remember to compile a routine that has been changed or that uses changed declarations.

The "make" is a software tool that maintains, updates, and regenerates groups of computer programs.

A programmer can easily forget

- Files that are dependent upon other files.
- Files that were modified recently.
- Files that need to be reprocessed or recompiled after a change in the source.
- The exact sequence of operations needed to make an exercise a new version of the program.

The many activities of program development and maintenance are made simpler by the **make** program.

The **make** program provides a method for maintaining up-to-date versions of programs that result from many operations on a number of files. The **make** program can keep track of the sequence of commands that create certain files and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of a program, the **make** command creates the proper files simply, correctly, and with a minimum amount of effort. The **make** program also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

The basic operation of **make** is to

- Find the name of the needed target file in the description.
- Ensure that all of the files on which it depends exit and are up to date.
- Create the target file if it has not been modified since its generators were modified.

The descriptor file really defines the graph of dependencies. The **make** program determines the necessary work by performing a depth-first search of the graph of dependencies.

If the information on interfile dependencies and command sequences is stored in a file, the simple command

```
make
```

is frequently sufficient to update the interesting files regardless of the number edited since the last **make**. In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the **make** command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think - edit - **make** - test ...

The **make** program is most useful for medium-sized programming projects. The **make** program does not solve the problems of maintaining multiple source versions or of describing huge programs.

As an example of the use of **make**, the description file used to maintain the **make** command is given. The code for **make** is spread over a number of C language source files and a Yacc grammar. The description file contains:

```
# Description file for the Make command

p = lp
FILES = Makefile version.c defs main.c doname.c misc.c
        files.c dosys.c gram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o
          dosys.o gram.o
LIBES= -lS
LINT = lint -p
CFLAGS = -O

make:  $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        size make

$(OBJECTS):  defs
gram.o: lex.c

cleanup:
        -rm  *.o gram.c
        -du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make

print:  $(FILES)        # print recently changed files
        pr $? | $P
        touch print
```

```
test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp | grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap

lint : dosys.c doname.c files.c main.c misc.c version.c
       gram.c
       $(LINT) dosys.c doname.c files.c main.c misc.c
       version.c gram.c

arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

The **make** program usually prints out each command before issuing it.

The following output results from typing the simple command **make** in a directory containing only the source and description files:

```
cc -O -c version.c
cc -O -c main.c
cc -O -c doname.c
cc -O -c misc.c
cc -O -c files.c
cc -O -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -O -c gram.c
cc version.o main.o doname.o misc.o files.o dosys.o
   gram.o -lS -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the description file, **make** found them using its suffix rules and issued the needed commands. The string of digits results from the **size make** command. The printing of the command line itself was suppressed by an @ sign. The @ sign on the **size** command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files changed since the last **make print** command. A zero-length file *print* is maintained to keep track of the time of the printing. The **$?** macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the **P** macro as follows:

```
make print " P=  cat >zap"
```

BASIC FEATURES

The basic operation of **make** is to update a target file by ensuring that all of the files on which the target file depends exist and are up to date. The target file is created if it has not been modified since the dependents were modified. The **make** program does a depth-first search of the graph of dependencies. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, consider a simple example in which a program named *prog* is made by compiling and loading three C language files *x.c*, *y.c*, and *z.c* with the **lS** library. By convention, the output of the C language compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c* and *y.c* have the line

```
#include " defs"
```

The following text describes the relationships and operations:

```
prog : x.o y.o z.o
       cc x.o y.o z.o -lS -o prog

x.o y.o :  defs
```

If this information were stored in a file named *makefile*, the command

```
make
```

would perform the operations needed to recreate *prog* after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

The **make** program operates using the following three sources of information:

- A user-supplied description file
- File names and "last-modified" times from the file system
- Built-in rules to bridge some of the gaps.

In the example, the first line states that *prog* depends on three "*.o*" files. Once these object files are current, the second line describes how to load them to create *prog*. The third line states that *x.o* and *y.o* depend on the file *defs*. From the file system, **make** discovers that there are three "*.c*" files corresponding to the needed "*.o*" files and uses built-in information on how to generate an object from a source file (i.e., issue a "cc -c" command).

By not taking advantage of **make**'s innate knowledge, the following longer descriptive file results.

```
prog : x.o y.o z.o
       cc x.o y.o z.o -lS  -o  prog
x.o : x.c defs
       cc -c  x.c
y.o : y.c defs
       cc -c  y.c
z.o : z.c
       cc -c  z.c
```

If none of the source or object files have changed since the last time *prog* was made, all of the files are current, and the command

```
make
```


announces this fact and stops. If, however, the *defs* file has been edited, *x.c* and *y.c* (but not *z.c*) is recompiled; and then *prog* is created from the new "*.o*" files. If only the file *y.c* had changed, only it is recompiled; but it is still necessary to reload *prog*. If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, the file's time of last modification is used in further decisions. If the file does not exist after the commands are executed, the current time is used in making further decisions. A method, often useful to programmers, is to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros. Thus, an entry "save" might be included to copy a certain set of files, or an entry "cleanup" might be used to throw away unneeded intermediate files. In other cases, one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

The **make** program has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign. Macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical. A $$ is a dollar sign.

The $*, $@, $?, and $< are four special macros which change values during the execution of the command. (These four macros are described in the part "DESCRIPTION FILES AND SUBSTITUTIONS".) The following fragment shows assignment and use of some macros:

```
OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
      cc $(OBJECTS) $(LIBES) -o prog
 . . .
```

The **make** command loads the three object files with the **lS** library. The command

```
make " LIBES= -ll -lS"
```

loads them with both the Lex (-ll) and the standard (-lS) libraries since macro definitions on the command line override definitions in the description. Remember to quote arguments with embedded blanks in UNIX software commands.

DESCRIPTION FILES AND SUBSTITUTIONS

A description file contains the following information:

- macro definitions
- dependency information
- executable commands.

The comment convention is that a sharp (#) and all characters on the same line after a sharp are ignored. Blank lines and lines beginning with a sharp (#) are totally ignored. If a noncomment line is too long, the line can be continued by using a backslash. If the last character of a line is a backslash, then the backslash, the new line, and all following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped). The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lS
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as the macro's value.

Macro definitions may also appear on the **make** command line while other lines give information about target files. The general form of an entry is

```
target1 [target2 . .] :[:] [dependent1 . .] [; commands] [# . .]
[(tab) commands] [# . . .]
 . . .
```

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. Shell metacharacters such as "*" and "?" are expanded. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line. A command is any string of characters not including a sharp (#) except when the sharp is in quotes or not including a new line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type. For the usual single-colon case, a command sequence may be associated with at most one dependency line. If the target is out of date with any of the dependents on any of the lines and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default creation rule may be invoked. In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in the silent mode or if the command line begins with an @ sign. **Make** normally stops if any command signals an error by returning a nonzero error code. Errors are ignored if the **–i** flags have been specified on the **make** command line, if the fake target name ".IGNORE" appears in the description file, or if the command string in the description file begins with a hyphen. Some UNIX software commands return meaningless status. Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (e.g., **cd** and shell control commands) that have meaning only within a single shell process. These results are forgotten before the next line is executed.

Before issuing any command, certain internally maintained macros are set. The $@ macro is set to the full target name of the current target. The $@ macro is evaluated only for explicitly named dependencies. The $? macro is set to the string of names that were found to be younger than the target. The $? macro is evaluated when explicit rules from the *makefile* are evaluated. If the command was generated by an implicit rule, the $< macro is the name of the related file that caused the action; and the $* macro is the prefix shared by the current and the dependent file names. 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 there is no such name, **make** prints a message and stops.

COMMAND USAGE

The **make** command takes macro definitions, flags, description file names, and target file names as arguments in the form:

make [flags] [macro definitions] [targets]

The following summary of command operations explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files. Next, the flag arguments are examined. The permissible flags are as follows:

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

–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 "@" sign are printed.

–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 nonzero status code depending on whether the target file is or is not up to date.

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

−d Debug mode. Print out detailed information on files and times examined.

−f Description file name. The next argument is assumed to be the name of a description file. A file name of "-" denotes the standard input. If there are no "*-f*" arguments, the file named *makefile* or *Makefile* in the current directory is read. The contents of the description files override the built-in rules if they are present.

Finally, the remaining arguments are assumed to be the names of targets to be made, and the arguments are done in left-to-right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

SUFFIXES AND TRANSFORMATION RULES

The **make** program does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the **−r** flag is used, the internal table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES". The **make** program searches for a file with any of the suffixes on the list. If such a file exists and if there is a transformation rule for that combination, **make** transforms a file with one suffix into a file with another suffix. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a *.r* file to a *.o* file is thus **.r.o**. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule **.r.o** is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made; and the macro $< is the name of the dependent that caused the action.

The order of the suffix list is significant since the list is scanned from left to right. The first name formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can add an entry for ".SUFFIXES" in his own description file. The dependents are added to the usual list. A ".SUFFIXES" line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed. The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC = yacc
YACCR = yacc -r
YACCE = yacc -e
YFLAGS =
LEX = lex
LFLAGS =
CC = cc
AS = as -
CFLAGS =
RC = ec
RFLAGS =
EC = ec
EFLAGS =
FFlags =
.c.o :
        $(CC) $(CFLAGS) -c  $<
.e.o .r.o .f.o :
        $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c  $<
.s.o :
        $(AS) -o  $@ $<
.y.o :
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c  y.tab.c
        rm y.tab.c
        mv y.tab.o $@
.y.c :
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@
```

IMPLICIT RULES

The **make** program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list is as follows:

.o	Object file
.c	C source file
.e	Efl source file
.r	Ratfor source file
.f	Fortran source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.ye	Yacc-Efl source grammar
.l	Lex source grammar.

Figure 2-1 summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.

If the file *x.o* were needed and there were an *x.c* in the description or directory, the *x.o* file would be compiled. If there were also an *x.l*, that grammar would be run through Lex before compiling the result. However, if there were no *x.c* but there were an *x.l*, **make** would discard the intermediate C language file and use the direct link as shown in Figure 2-1.

It is possible to change the names of some of the compilers used in the default or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros **AS**, **CC**, **RC**, **EC**, **YACC**, **YACCR**, **YACCE**, and **LEX**. The command

```
make CC=newcc
```

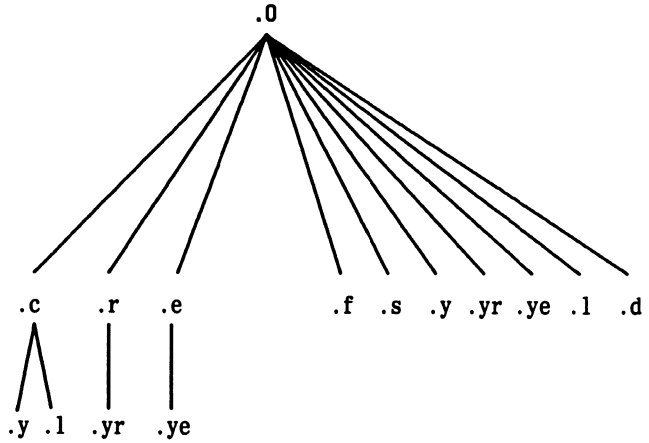



Figure 2-1. Summary of Default Transformation Path

will cause the **newcc** command to be used instead of the usual C language compiler. The macros **CFLAGS**, **RFLAGS**, **EFLAGS**, **YFLAGS**, and **LFLAGS** may be set to cause these commands to be issued with optional flags. Thus

```
make " CFLAGS=-O"
```

causes the optimizing C language compiler to be used.

SUGGESTIONS AND WARNINGS

The most common difficulties arise from **make**'s specific meaning of dependency. If file *x.c* has a "#include " defs" " line, then the object file *x.o* depends on **defs**; the source file *x.c* does not. If **defs** is changed, nothing is done to the file *x.c* while file *x.o* must be recreated.

To discover what **make** would do, the **−n** option is very useful. The command

```
make -n
```

orders **make** to print out the commands which **make** would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be mild in character (e.g., adding a new definition to an include file), the **−t** (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the command

```
make -ts
```

("touch silently") causes the relevant files to appear up to date. Obvious care is necessary since this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

The debugging flag (**−d**) causes **make** to print out a very detailed description of what it is doing including the file times. The output is verbose and recommended only as a last resort.

Chapter 3

AUGMENTED VERSION OF make

PAGE

Chapter 3

AUGMENTED VERSION OF make

GENERAL

This section describes an augmented version of the **make** command of the UNIX operating system. The augmented version is upward compatible with the old version. This section describes and gives examples of only the additional features. Further possible developments for **make** are also discussed. Some justification will be given for the chosen implementation, and examples will demonstrate the additional features.

The **make** command is an excellent program administrative tool used extensively in at least one project for over 2 years. However, **make** had the following shortcomings:

- Handling of libraries was tedious.
- Handling of the Source Code Control System (SCCS) file name format was difficult or impossible.
- Environment variables were completely ignored by **make**.
- The general lack of ability to maintain files in a remote directory.

These shortcomings hindered large scale use of **make** as a program support tool.

The AUGMENTED VERSION OF make is modified to handle the above problems. The additional features are within the original syntactic framework of **make** and few if any new syntactical entities are introduced. A notable exception is the *include* file capability. Further, most of the additions result in a "Don't know how to make ..." message from the old version of **make**.

The following paragraphs describe with examples the additional features of the **make** program. In general, the examples are taken from existing *makefiles*. Also, the illustrations are examples of working *makefiles*.

THE ENVIRONMENT VARIABLES

Environment variables are read and added to the macro definitions each time **make** executes. Precedence is a prime consideration in doing this properly. The following describes **make**'s interaction with the environment. A new macro, **MAKEFLAGS**, is maintained by **make**. The new macro is defined as the collection of all input flag arguments into a string (without minus signs). The new macro is exported and thus accessible to further invocations of **make**. Command line flags and assignments in the *makefile* update **MAKEFLAGS**. Thus, to describe how the environment interacts with **make**, the **MAKEFLAGS** macro (environment variable) must be considered.

When executed, **make** assigns macro definitions in the following order:

1. Read the **MAKEFLAGS** environment variable. If it is not present or null, the internal **make** variable **MAKEFLAGS** is set to the null string. Otherwise, each letter in **MAKEFLAGS** is assumed to be an input flag argument and is processed as such. (The only exceptions are the **−f**, **−p**, and **−r** flags.)

2. Read and set the input flags from the command line. The command line adds to the previous settings from the **MAKEFLAGS** environment variable.

3. Read macro definitions from the command line. These are made *not resettable*. Thus, any further assignments to these names are ignored.

4. Read the internal list of macro definitions. These are found in the file *rules.c* of the source for **make**. Figure 3-1 contains the complete makefile that represents the internally defined macros and rules of the current version of **make**. Thus, if **make −r ...** is typed and a *makefile* includes the *makefile* in Figure 3-1, the results would be identical to excluding the **−r** option and the *include* line in the *makefile*. The Figure 3-1 output can be reproduced by the following:

```
make -fp - < /dev/null 2>/dev/null
```

The output appears on the standard output.
They give default definitions for the C language compiler (CC=cc), the assembler (AS=as), etc.

5. Read the environment. The environment variables are treated as macro definitions and marked as *exported* (in the shell sense). However, since **MAKEFLAGS*** is not an internally defined variable (in *rules.c*), this has the effect of doing the same assignment twice. The exception to this is when **MAKEFLAGS** is assigned on the command line. (The reason it was read previously was to turn the debug flag on before anything else was done.)

6. Read the *makefile*(*s*). The assignments in the *makefile(s)* overrides the environment. This order is chosen so that when a *makefile* is read and executed, you know what to expect. That is, you get what is seen unless the **−e** flag is used. The **−e** is an additional command line flag which tells **make** to have the environment override the *makefile* assignments. Thus, if **make −e ...** is typed, the variables in the environment override the definitions in the *makefile*†. Also **MAKEFLAGS** override the environment if assigned. This is useful for further invocations of **make** from the current *makefile*.

---

* **:MAKEFLAGS** are read and set again.

† There is no way to override the command line assignments.

```
#               LIST OF SUFFIXES

.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~
        .sh .sh~ .h .h~

#               PRESET VARIABLES

MAKE=make
YACC=yacc
YFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
CC=cc
CFLAGS=-o
AS=as
ASFLAGS=
GET=get
GFLAGS=
```

Figure 3-1. Example of Internal Definitions (Sheet 1 of 4)


```
#               SINGLE SUFFIX RULES

.c:
        $(CC) -n -o $< -o $@

.c~:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) -n -o $* .c -o $*
        -rm -f $*.c

.sh:
        cp $< @

.sh~:
        $(GET) &(GFLAGS) -p $< > .sh
        cp $* .sh $*
        -rm -f $* .sh

#               DOUBLE SUFFIX RULES

.c.o:
        $(CC) $(CFLAGs) -c $<

.c~.o:
```

Figure 3-1. Example of Internal Definitions (Sheet 2 of 4)

```
	$(GET) $(CFLAGS) -p $< > $*.c
	$(CC) $(CFLAGS) -c $*.c
	-rm -f $*.c
.c~ .c:
	$(GET) $(GFLAGS) -p $< >$*.c
.s.o:
	$(AS) $(ASFLAGS) -o $@ $<
.s~ .o:
	$(GET) $(GFLAGS) -p $< > $*.s
	$(AS) $(ASFLAGS) -o $* .o $* .s
	-rm -f $*.s
.y.o:
	$(YACC) $(YFLAGS) $<
	$(CC) $(CFLAGS) -c y.tab.c
	rm y.tab.o$@
.y~.o:
	$(GET) $(GFLAG) -p $< > $*.y
	$(YACC) $(YFLAGS) $*.y
	$(CC) $(CFLAG) -c y.tab.c
	rm -f y.tab $*.y
	mv y.tab.o $*.o
.l.o:
	$(LEX) $(LFLAGS) $<
	$(CC) $(CFLAGS) -c lex.yy.c
	rm lex.yy.c
	mv  lex.yy.o $@
```

Figure 3-1. Example of Internal Definitions (Sheet 3 of 4)


```
.l~ .o:
	$(GET) $(GFLAGS) -p $< > $*.l
	$(LEX) $(GFLAG) $*.l
	$(CC) $(CFLAGS) -c lex.yy.c
	rm -f lex.yy.c $*.l
	mv lex.yy.o $*.o
.y .c:
	$(YACC) $(YFLAGS) $<
	mv y.tab.c $@
.y~.c:
	$(GET) $( GFLAGS) -p $< > $*.y
	$(YACC) $(YFLAGS) $*.y
	mv -f $*.c
	-rm -f $*.y
.l.c:
	$(LEX) $<
	mv lex.yy.c$@
.c.a:
	$(CC) -c $(FLAGS) $<
	ar rv $@ $*.o
	rm -f $*.o
.c~.a:
	$(GET) $(GFLAGS) -p $< > $*.c
	$(CC) -c $(CFLAGS) $*.c
	ar rv $@ $*.o
.s~.a:
	$(GET) $(GFLAGS) -p $< > $*.s
	$(AS) $(ASFLAGS) -o $*.o $*.s
	ar rv $@ $*.o
	-rm -f $*.[so]
.h~.h
	$(GET) $(GFLAGS) -p $< > $*.h
```

Figure 3-1. Example of Internal Definitions (Sheet 4 of 4)

It may be clearer to list the precedence of assignments. Thus, in order from least binding to most binding, the precedence of assignments is as follows:

1. internal definitions (from *rules.c*)
2. environment
3. *makefile(s)*
4. command line.

The **–e** flag has the effect of changing the order to:

1. internal definitions (from *rules.c*)
2. *makefile(s)*
3. environment
4. command line.

This order is general enough to allow a programmer to define a *makefile* or set of *makefiles* whose parameters are dynamically definable.

RECURSIVE MAKEFILES

Another feature was added to **make** concerning the environment and recursive invocations. If the sequence "$(MAKE)" appears anywhere in a shell command line, the line is executed even if the **–n** flag is set. Since the **–n** flag is exported across invocations of **make** (through the **MAKEFLAGS** variable), the only thing that actually gets executed is the **make** command itself. This feature is useful when a hierarchy of *makefile(s)* describes a set of software subsystems. For testing purposes, **make –n ...** can be executed and everything that would have been done will get printed out including output from lower level invocations of **make**.

FORMAT OF SHELL COMMANDS WITHIN make

The **make** program remembers embedded newlines and tabs in shell command sequences. Thus, if the programmer puts a *for* loop in the makefile with indentation, when **make** prints it out, it retains the indentation and backslashes. The output can still be piped to the shell and is readable. This is obviously a cosmetic change; no new function is gained.

ARCHIVE LIBRARIES

The **make** program has an improved interface to archive libraries. Due to a lack of documentation, most people are probably not aware of the current syntax of addressing members of archive libraries. The previous version of **make** allows a user to name a member of a library in the following manner:

```
lib(object.o)
   or
lib((_localtime))
```

where the second method actually refers to an entry point of an object file within the library. (**Make** looks through the library, locates the entry point, and translates it to the correct object file name.)

To use this procedure to maintain an archive library, the following type of *makefile* is required:

```
lib:: lib(ctime.o)
      $(CC) -c -O ctime.c
      ar rv lib ctime.o
      rm ctime.o
lib:: lib(fopen.o)
      $(CC) -c -O fopen.c
      ar rv lib fopen.o
      rm fopen.o
...and so on for each object ...
```

This is tedious and error prone. Obviously, the command sequences for adding a C language file to a library are the same for each invocation; the file name being the only difference each time. (This is true in most cases.)

The current version gives the user access to a rule for building libraries. The handle for the rule is the ".a" suffix. Thus, a ".c.a" rule is the rule for compiling a C language source file, adding it to the library, and removing the ".o" cadaver. Similarly, the ".y.a", the ".s.a", and the ".l.a" rules rebuild YACC, assembler, and LEX files, respectively. The current archive rules defined internally are ".c.a", ".c{.a", and ".s{.a". [The tilde (~) syntax will be described shortly.] The user may define in makefile other rules needed.

The above 2-member library is then maintained with the following shorter makefile:

```
lib:    lib(ctime.o) lib(fopen.o)
        echo lib up-to-date.
```

The internal rules are already defined to complete the preceding library maintenance. The actual ".c.a" rules are as follows:

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
```

Thus, the $@ macro is the ".a" target (lib); the $< and $* macros are set to the out-of-date C language file; and the file name scans the suffix, respectively (*ctime.c* and *ctime*). The $< macro (in the preceding rule) could have been changed to **$*.c**.

It might be useful to go into some detail about exactly what **make** does when it sees the construction

```
lib:    lib(ctime.o)
        @echo lib up-to-date
```


Assume the object in the library is out-of date with respect to *ctime.c*. Also, there is no *ctime.o* file.

1. Do *lib*.
2. To do *lib*, do each dependent of *lib*.
3. Do *lib*(*ctime.o*).
4. To do *lib*(*ctime.o*), do each dependent of *lib*(*ctime.o*). (There are none.)
5. Use internal rules to try to build *lib*(*ctime.o*). (There is no explicit rule.) Note that *lib*(*ctime.o*) has a parenthesis in the name to identify the target suffix as ".a". This is the key. There is no explicit ".a" at the end of the *lib* library name. The parenthesis forces the ".a" suffix. In this sense, the ".a" is hard wired into **make**.
6. Break the name *lib*(*ctime.o*) up into *lib* and *ctime.o*. Define two macros, $@ (=*lib*) and $* (=*ctime*).
7. Look for a rule ".X.a" and a file **$*.X**. The first ".X" (in the .SUFFIXES list) which fulfills these conditions is ".c" so the rule is ".c.a", and the file is *ctime.c*. Set $< to be *ctime.c* and execute the rule. In fact, **make** must then do *ctime.c*. However, the search of the current directory yields no other candidates, and the search ends.
8. The library has been updated. Do the rule associated with the "lib:" dependency; namely:

```
echo lib up-to-date
```

It should be noted that to let *ctime.o* have dependencies, the following syntax is required:

```
lib(ctime.o):   $(INCDIR)/stdio.h
```

Thus, explicit references to *.o* files are unnecessary. There is also a new macro for referencing the archive member name when this form

is used. The $% macro is evaluated each time $@ is evaluated. If there is no current archive member, $% is null. If an archive member exists, then $% evaluates to the expression between the parenthesis.

An example *makefile* for a larger library is given in Figure 3-2.

```
#                @(#)/usr/src/cmd/make/make.tm 3.2
LIB ==lsxlib
PR==lp
INSDIR == /rl/flop0/
INS == eval
lsx:            $(LIB) low.o mch.o
                ld -x low.o mch.o $(LIB)
                mv a.out lsx
                @size lsx
#               Here, $(INS) as either " ." or " eval".
lsx:
                $(INS)'cp lsx $(INSDIR)lsx . .
                    strip $(INSDIR)lsx . .
                    ls -l $(INSDIR)lsx'
print:
                $(PR) header.slow.smch.s*.h*.c Makefile
```

Figure 3-2. Example of Library Makefile (Sheet 1 of 3)

```
$(LIB):
        $(LIB)(CLOCK.o)
        $(LIB)(main.o)
        $(LIB)(tty.o)
        $(LIB)(trap.o)
        $(LIB)(sysent.o)
        $(LIB)(sys2.o)
        $(LIB)(syn3.o)
        $(LIB)(syn4.o)
        $(LIB)(sys1.o)
        $(LIB)(sig.o)
        $(LIB)(fio.o)
        $(LIB)(kl.o)
        $(LIB)(alloc.o)
        $(LIB)(nami.o)
        $(LIB)(iget.o)
        $(LIB)(rdwri.o)
        $(LIB)(subr.o)
```

Figure 3-2. Example of Library Makefile (Sheet 2 of 3)

```
        $(LIB)(bio.o)
        $(LIB)(decfd.o)
        $(LIB)(sip.o)
        $(LIB)(space.o)
        $(LIB)(puts.o)
        @echo $(LIB) now up to date.
.s.o:
        as -o $*.o header.s $*.s
.o.a:
        ar rv $@ $<
        rm -f $<
.s.a:
        as -o $*.o header.s $*.s
        ar rv $@ $*.o
        rm -f $*.o
.PRECIOUS:$(LIB)
```

Figure 3-2. Example of Library Makefile (Sheet 3 of 3)

The reader will note also that there are no lingering "*.o" files left around. The result is a library maintained directly from the source files (or more generally from the SCCS files).

SOURCE CODE CONTROL SYSTEM FILE NAMES: THE TILDE

The syntax of **make** does not directly permit referencing of prefixes. For most types of files on UNIX operating system machines, this is acceptable since nearly everyone uses a suffix to distinguish different types of files. The SCCS files are the exception. Here, "s." precedes the file name part of the complete pathname.

To allow **make** easy access to the prefix "s." requires either a redefinition of the rule naming syntax of **make** or a trick. The trick is to use the tilde (~) as an identifier of SCCS files. Hence, ".c~.o" refers to the rule which transforms an SCCS C language source file into an object. Specifically, the internal rule is

```
.c~.o:
        $(GET) $(GFLAGS) -p  $< > $*.c
        $(CC) $(CFLAGS) -c  $*.c
        -rm  -f  $*.c
```

Thus, the tilde appended to any suffix transforms the file search into an SCCS file name search with the actual suffix named by the dot and all characters up to (but not including) the tilde.

The following SCCS suffixes are internally defined:

```
.c~
.y~
.s~
.sh~
.h~
```

The following rules involving SCCS transformations are internally defined:

```
.c~:
.sh~:
.c~.o:
.s~.o:
.y~.o:
.l~.o:
.y~.c:
.c~.a:
.s~.a:
.h~.h:
```

Obviously, the user can define other rules and suffixes which may prove useful. The tilde gives him a handle on the SCCS file name format so that this is possible.

THE NULL SUFFIX

In the UNIX system source code, there are many commands which consist of a single source file. It was wasteful to maintain an object of such files for **make**. The current implementation supports single suffix rules (a null suffix). Thus, to maintain the program *cat*, a rule in the *makefile* of the following form is needed:

```
.c:
        $(CC) -n -O $< -o $@
```

In fact, this ".c:" rule is internally defined so no *makefile* is necessary at all. The user only needs to type

```
make cat dd echo date
```

(these are notable single file programs) and all four C language source files are passed through the above shell command line associated with the ".c:" rule. The internally defined single suffix rules are

```
.c:
.c~:
.sh:
.sh~:
```

Others may be added in the *makefile* by the user.

INCLUDE FILES

The **make** program has an include file capability. 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 string is assumed to be a file name which the current invocation of **make** will read. The file descriptors are stacked for reading *include* files so that no more than about 16 levels of nested *includes* are supported.

INVISIBLE SCCS MAKEFILES

The SCCS *makefiles* are invisible to **make**. That is, if **make** is typed and only a file named *s.makefile* exists, **make** will do a **get** on the file, then read and remove the file. Using the **–f**, **make** will get, read, and remove arguments and *include* files.

DYNAMIC DEPENDENCY PARAMETERS

A new dependency parameter has been defined. The parameter has meaning only on the dependency line in a makefile. The $$@ refers to the current "thing" to the left of the colon (which is $@). Also the form **$$(@F)** exists which allows access to the file part of $@. Thus, in the following:

```
cat:    $$@.c
```

the dependency is translated at execution time to the string "cat.c". This is useful for building a large number of executable files, each of which has only one source file. For instance, the UNIX software command directory could have a *makefile* like:

```
CMDS = cat dd echo date cc cmp comm ar ld chown

$(CMDS):        $$@.c
        $(CC) -O $? -o $@
```

Obviously, this is a subset of all the single file programs. For multiple file programs, a directory is usually allocated and a separate *makefile* is made. For any particular file that has a peculiar compilation procedure, a specific entry must be made in the *makefile.*

The second useful form of the dependency parameter is **$$(@F)**. It represents the file name part of $$@. Again, it is evaluated at execution time. Its usefulness becomes evident when trying to maintain the */usr/include* directory from a makefile in the */usr/src/head* directory. Thus, the */usr/src/head/makefile* would look like

```
INCDIR = /usr/include

INCLUDES = \
        $(INCDIR)/stdio.h \
        $(INCDIR)/pwd.h \
        $(INCIDR)/dir.h \
        $(INCDIR)/a.out.h

$(INCLUDES): $$(@F)
        cp $? $@
        chmod 0444 $@
```

This would completely maintain the */usr/include* directory whenever one of the above files in */usr/src/head* was updated.

EXTENSIONS OF $*, $@, AND $<

The internally generated macros $*, $@, and $< are useful generic terms for current targets and out-of-date relatives. To this list has been added the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)**. The "D" refers to the directory part of the single letter macro. The "F" refers to the file name part of the single letter macro. These additions are useful when building hierarchical makefiles. They allow access to directory names for purposes of using the **cd** command of the shell. Thus, a shell command can be

```
cd $(<D); $(MAKE) $(<F)
```

An interesting example of the use of these features can be found in the set of *makefiles* in Figure 3-3. Each *makefile* is named "70.mk". The following command forces a complete rebuild of the operating system:

```
FRC=FRC make -f 70.mk
```

where the current directory is *ucb*. The FRC is a convention for *FoRC*ing **make** to completely rebuild a target starting from scratch.

OUTPUT TRANSLATIONS

Macros in shell commands can now be translated when evaluated. The form is as follows:

```
$(macro:string1=string2)
```

The meaning of **$(macro)** is evaluated. For each appearance of *string1* in the evaluated macro, *string2* is substituted. The meaning of finding *string1* in **$(macro)** is that the evaluated **$(macro)** is considered as a bunch of strings each delimited by white space (blanks or tabs). Thus, the occurrence of *string1* in **$(macro)** means that a regular expression of the following form has been found:

```
.*<string1>[TAB¦BLANK]
```

This particular form was chosen because **make** usually concerns itself with suffixes. A more general regular expression match could be implemented if the need arises. The usefulness of this type of translation occurs when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script which can handle all the C language programs (i.e., those files ending in ".c"). Thus, the following fragment optimizes the executions of **make** for maintaining an archive library:

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)c.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        ar rv $(LIB) $?
        rm $?
```

A dependency of the preceding form is necessary for each of the different types of source files (suffices) which define the archive library. These translations are added in an effort to make more general use of the wealth of information which **make** generates.

Bell Laboratories

ect: An Augmented Version of Make
Charge Case 49579-210
File Case 40320-1

date: July 1, 1979

from: E. G. Bradford
CB 5255
1C-249 x2804

TM 79-5255-1
5255-790701.01MF

ABSTRACT

This paper describes an augmented version of the **make** command supplied with UNIX/TS. With one debatable exception, this version is completely upward compatible with the UNIX/TS version. In this paper, I describe and give examples only of additional features. The reader is assumed to have read or have available the original **make** paper by S. I. Feldman.[1] Further developments for **make** are also discussed.

MEMORANDUM FOR FILE

1. INTRODUCTION

This paper will describe in some detail an augmented version of the **make** program now running on the Columbus operating systems group UNIX machine. I will give some justification for the chosen implementation and describe with examples the additional features.

2. MOTIVATION FOR THE CURRENT IMPLEMENTATION

The **make** program was originally written for personal use by S. I. Feldman. However, it became popular on the research UNIX machine and a more formal version was built and installed for general use. For the purpose of maintaining executable programs in the Center 127 environment, it has served this purpose well. Further developments of **make** have not been necessary and thus have not been done.

In Columbus **make** was perceived as an excellent program administrative tool and has been used extensively in at least one project (NOCS) for over two years. However, **make** had many shortcomings: handling of libraries was tedious; handling of the SCCS filename format was difficult or impossible; environment variables are completely ignored by **make**; and the general lack of ability to maintain files in a remote directory. These shortcomings hindered large scale use of **make** as a program support tool.

1. Feldman, S. I., MAKE, A Program for Maintaining Computer Programs, Computing Science Technical Report Number 57

There were at least two avenues for solving the above problems. The first was a complete redesign. This would probably mean a new syntax and of necessity force new makefiles to be incompatible with old ones. The advantages however would be a more general implementation that would be *growable.* This point of view was not chosen because of the compatibility problem. The second and more tame point of view was to modify the current implementation to handle the problems above. This point of view had the advantage that if done carefully it could be completely upward compatible. It was this second avenue which was chosen.

The additional features are within the original syntactic framework of **make** and few if any new syntactical entities have been introduced. A notable exception is the *include* file capability. Further, most of the additions result in a "Don't know how to make ..." message from the old version of **make.**

3. THE ADDITIONAL FEATURES

The following paragraphs describe with examples the additional features of the **make** program. In general, the examples are taken from existing *makefiles.* Also, the appendices are working *makefiles.*

3.1 The Environment Variables

Environment variables are read and added to the macro definitions each time **make** executes. Precedence is a prime consideration in doing this properly. Thus, if the environment variable CC is set to occ, does it override the command line? Does it override the definition in the makefile? To answer these questions I need to describe the order in which **make** does the macro assignments.

First, a new macro, MAKEFLAGS, must be described. MAKEFLAGS is maintained by **make.** It It is defined as the collection of all input flag arguments into a string (without the minus sign). It is exported, and thus accessible to further invocations of **make.** Command line flags and assignments in the "makefile" update MAKEFLAGS. Thus, to describe how the environment interacts with **make,** we also need to consider the MAKEFLAGS macro (environment variable).

When executed **make** assigns macro definitions in the following order:

1. read the MAKEFLAGS environment variable. If it is not present or null, the internal **make** variable MAKEFLAGS is set to the null string. Otherwise, each letter in MAKEFLAGS is assumed to be an input flag argument and is processed as such. (The only exceptions are the "-f", "-p", and "-r" flags.)
2. read and set the input flags from the command line. The command line adds to the previous settings from the MAKEFLAGS environment variable.
3. read macro definitions from the command line. These are made *not resettable.* Thus any further assignments to these names are ignored.
4. read the internal list of macro definitions. These are found in the file *files.c* of the source for **make.** (See Appendix A for the complete makefile which represents the internally defined macros and rules.) They give default definitions for the C compiler (CC=cc), the assembler (AS=as), etc.
5. read the environment. The environment variables are treated as macro definitions and marked as *exported* (in the shell sense). Note, MAKEFLAGS will get read again and set again. However, since it is not an internally defined variable (in *files.c*), this has the effect of doing the same assignment twice. The exception to this is when MAKEFLAGS is assigned on the command line. (The reason it was read previously, was to be able to turn the debug flag on before anything else was done.)

6. read the *makefile(s)*. The assignments in the *makefile(s)* will override the environment. This order was chosen so when one reads a makefile and executes **make** one knows what to expect. That is, one gets what one sees unless the "-e" flag is used. The "-e" is an additional command line flag which tells **make** to have the environment override the *makefile* assignments. Thus if **make -e ...** is typed, the variables in the environment override the definitions in the *makefile*. (Note, there is no way to override the command line assignments.) Also note that if **MAKEFLAGS** is assigned it will override the environment. (This would be useful for further invocations of **make** from the current "makefile".)

This description may be hard to follow. No doubt it is. It might be more useful to list the precedence of assignments. Thus, in order from least binding to most binding, we have:

1. internal definitions (from *files.c*)
2. environment
3. "makefile(s)"
4. command line

The "-e" flag has the effect of changing the order to:

1. internal definitions (from *files.c*)
2. "makefile(s)"
3. environment
4. command line

This ordering is general enough to allow a programmer to define a "makefile" or set of "makefiles" whose parameters are dynamically definable.

3.2 Recursive Makefiles

One other useful feature was added to **make** concerning the environment and recursive invocations. If the sequence "$(MAKE)" appears anywhere in a shell command line, the line will be executed even if the "-n" flag is set. Since the "-n" flag is exported across invocations of **make**, (through the **MAKEFLAGS** variable) the only thing which will actually get executed is the **make** command itself. This feature is useful when a hierarchy of *makefile(s)* describes a set of software subsystems. For testing purposes, **make -n ...** can be executed and everything that would have been done will get printed out; including output from lower level invocations of **make**.

3.3 Format of Shell commands within make

Make remembers embedded newlines and tabs in shell command sequences. Thus, if the programmer puts a *for* loop in the makefile with indentation, when **make** prints it out, it still has the indentation and the backslashes in it. The output is still pipe-able to the shell and is readable. This is obviously a cosmetic change; no new functionality is gained.

3.4 Archive Libraries

Make has an intelligent interface to archive libraries. Due to a lack of documentation, most people are probably not aware of the current syntax of addressing members of archive libraries. The UNIX/TS version allows a user to name a member of a library in the following manner:

```
lib(object.o)
```

or

```
lib((_localtime))
```

where the second method actually refers to an entry point of an object file within the library.

(Make looks through the library, locates the entry point and translates it to the correct object file name.)

To use the UNIX/TS make to maintain an archive library, the following type of **makefile** is required:

```
lib::   lib(ctime.o)
        $(CC) -c -O ctime.c
        ar rv lib ctime.o
        rm ctime.o
lib::   lib(fopen.o)
        $(CC) -c -O fopen.c
        ar rv lib fopen.o
        rm fopen.o
...and so on for each object ...
```

This is tedious and error prone. Obviously, the command sequences for adding a C file to a library are the same for each invocation, the filename being the only difference each time. (This is true in most cases.) Similarly for assembler and YACC and LEX files.

The current version gives the user access to a rule for building libraries. The "handle" for the rule is the ".a" suffix. Thus a ".c.a" rule is the rule for compiling a C source file and adding it to the library and removing the ".o" cadaver. Similarly, the ".y.a", the ".s.a" and the ".l.a" rules rebuild YACC, assembler, and LEX files respectively. The current archive rules defined internally are ".c.a", ".c~.a", and .s~.a". (The wiggle (~) syntax will be described shortly.) The user may define in his makefile any other rules he may need.

The above two-member library is then maintained with the following shorter makefile:

```
lib:    lib(ctime.o) lib(fopen.o)
        @echo lib up-to-date.
```

The internal rules are already defined to complete the preceding library maintenance. The actual ".c.a" rules is as follows:

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
```

Thus, the "$@" macro is the ".a" target (**lib**) and the "$<" and "$*" macros are set to the out-of-date C file and the filename sans suffix respectively (**ctime.c** and **ctime**). The "$<" macro (in the preceding rule) could have been changed to "$*.c".

It might be useful to go into some detail about exactly what **make** thinks about when it sees the construction

```
lib:    lib(ctime.o)
        @echo lib up-to-date
```

Assume the object in the library is out of date with respect to *ctime.c*. Also, there is no *ctime.o* file. To itself, **make** thinks

1. I must do *lib*.
2. To do *lib*, I must do each dependent of *lib*.

3. I must do *lib(ctime.o)*.
4. To do *lib(ctime.o)* I must do each dependent of *lib(ctime.o)*. (There are none).
5. Use my internal rules to try to build *lib(ctime.o)*. (There is no explicit rule.) Note that *lib(ctime.o)* has a parenthesis, '(', in the name so I identify the target suffix as ".a". (This is the key. There is no explicit ".a" at the end of the *lib* library name. The parenthesis forces the ".a" suffix.) In this sense, the ".a" is hardwired into **make**.
6. Since I am working on a ".a" suffix I must break the name *lib(ctime.o)* up into *lib* and *ctime.o*. I now define the two macros "$@" (=*lib*) and "$*" (=*ctime*).
7. Look for a rule ".X.a" and a file "$*.X". The first "X" (in the .SUFFIXES list) which fulfills these conditions is "c" so the rule is ".c.a" and the file is *ctime.c*. I set "$<" to be *ctime.c* and execute the rule. (In fact, **make** must then do "ctime.c". However, the search of the current directory yields no other candidates, whence, the search ends.)
8. The library has been updated. I must now do the rule associated with the "lib:" dependency; namely

```
echo lib up-to-date
```

It should be noted that to let *ctime.o* have dependencies the following syntax is required:

```
lib(ctime.o):    $(INCDIR)/stdio.h
```

Thus, explicit references to ".o" files are unnecessary. There is also a new macro for referencing the archive member name when this form is used. "$%" is evaluated each time "$@" is evaluated. If there is no current archive member, "$%" is null. If an archive member exists, then "$%" evaluates to the expression between the parenthesis.

An example makefile for a larger library is given in Appendix B. The reader will note also, that there are no *lingering* "*.o" files left around. The result is a library maintained directly from the source files (or more generally from the SCCS files!).

3.5 SCCS File Names – The Wiggle

The syntax of **make** does not directly permit referencing of prefixes. For most types of files on UNIX machines this is acceptable since nearly everyone uses a suffix to distinguish different types of files. SCCS files are the exception. Here, "s." precedes the filename part of the complete pathname.

To allow **make** easy access to the prefix "s." requires either a redefinition of the rule naming syntax of **make** or a *trick*. I used a trick. The trick is to use the wiggle (~) as an identifier of SCCS files. Hence, ".c~.o" refers to the rule which transforms an SCCS C source file into an object. Specifically, the internal rule is:

```
.c~.o:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
```

(The motivation for the "-p" flag associated with the $(GET) command above is obscure and debatable. For the purpose of this discussion we can assume the following apparently equivalent rule:

```
.c~.o:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
```

Suffice it to say, that when doing a generic build, the "make" should not fail because a file happens to be left out in the SCCS directory.)

Thus the wiggle appended to any suffix transforms the file search into an SCCS filename search with the actual suffix named by the dot and all characters up to (but not including) the wiggle.

The following SCCS suffixes are internally defined:

```
.c~
.y~
.s~
.sh~
.h~
```

The following rules involving SCCS transformations are internally defined:

```
.c~:
.sh~:
.c~.o:
.s~.o:
.y~.o:
.l~.o:
.y~.c:
.c~.a:
.s~.a:
.h~.h:
```

Obviously, the user can define other rules and suffixes which may prove useful. The *wiggle* gives him a handle on the SCCS filename format so that this is possible.

3.6 The Null Suffix

In the UNIX/TS source code, there are many commands which consist of a single source file. *cat.c*, *dd.c*, *echo.c*, and *date.c* are a few well known ones. It seemed a pity to maintain an object of such files for **make**'s pleasure. The current implementation supports single suffix rules, or if one prefers, a null suffix. Thus, to maintain the above files one needs a *makefile* of the following form:

```
.c:
        $(CC) -n -O $< -o $@
```

(In fact, this ".c:" rule is internally defined so no *makefile* is necessary at all!) One then need only type

```
make cat dd echo date
```

and all four C source files are passed through the above shell command line associated with the ".c:" rule. The internally defined single suffix rules are:

```
.c:
.c~:
.sh:
.sh~:
```

Others may be added in the *makefile* by the user.

3.7 Include Files

Make has an include file capability. 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 following string is assumed to be a file name which the current invocation of **make** will read. The file descriptors are stacked for reading *include* files so no more that about sixteen levels of nested includes is supported. Does that bother anyone?

3.8 Invisible SCCS *Makefiles*

SCCS *makefiles* are invisible to **make**. That is, if **make** is typed and only a file named *s.makefile* exists, **make** will *get(I)* it, read it and remove it. Likewise for "-f" arguments and *include* files.

3.9 Dynamic Dependency Parameters

A new dependency parameter has been defined. It has meaning only on the dependency line in a makefile. It is "$$@". "$$@" refers to the current "thing" at the left of the colon (which is "$@"). Also the form "$$(@F)" exists which allows access to the file part of "$@". Thus, in the following:

```
cat:    $$@.c
```

the dependency is translated at execution time to the string "cat.c". This is useful for building a whole raft of executable files, each of which has only one source file. For instance the UNIX/TS command directory would have a *makefile* like:

```
CMDS = cat dd echo date cc cmp comm ar ld chown

$(CMDS):        $$@.c
        $(CC) -O $? -o $@
```

Obviously, this is a subset of all the single file programs. For multiple file programs, usually a directory is allocated and a separate *makefile* is made. For any particular file which has a peculiar compilation procedure, a specific entry must be made in the *makefile*.

The second useful form of the dependency parameter is "$$(@F)". It represents the filename part of "$$@". Again, it is evaluated at execution time. Its usefulness shows up when trying to maintain the "/usr/include" directory from a makefile in the "/usr/src/head" directory. Thus the "/usr/src/head/makefile" would look like:

```
INCDIR = /usr/include

INCLUDES = \
        $(INCDIR)/stdio.h \
        $(INCDIR)/pwd.h \
        $(INCIDR)/dir.h \
        $(INCDIR)/a.out.h
```

```
$(INCLUDES): $$(@F)
        cp $? $@
        chmod 0444 $@
```

would completely maintain the "/usr/include" directory whenever one of the above files in "/usr/src/head" was updated.

3.10 Extensions of $*, $@, and $<

The internally generated macros "$*", "$@", and "$<" are useful generic terms for current targets and out-of-date relatives. To this list has been added the following related macros: "$(@D)", "$(@F)", "$(*D)", "$(*F)", "$(<D)", and "$(<F)". The "D" refers to the directory part of the single letter macro. The "F" refers to the filename part of the single letter macro. These additions are useful when building hierarchical makefiles. They allow access to directory names for purposes of using the 'cd' command of the shell. Thus, a shell command can be:

```
cd $(<D); $(MAKE) $(<F)
```

An interesting example of the use of these features can be found in the set of *makefiles* which maintain the Columbus UNIX operating system. They may be seen in Appendix C.

3.11 Output Translations

Macros in shell commands can now be translated when evaluated. The form is as follows:

```
$(macro:string1=string2)
```

The meaning is as follows: $(macro) is evaluated. For each appearance of string1 in the evaluated macro string2 is substituted. The meaning of finding string1 in $(macro) is that the evaluated $(macro) is considered as a bunch of strings each delimited by whitespace (blanks or tabs). Thus the occurrence of string1 in $(macro) means that a regular expression of the following form has been found:

```
.*<string1>[TAB|BLANK]
```

This particular form was chosen because **make** usually concerns itself with suffixes. A more general regular expression match could be implemented if the need arises. The usefulness of this type of translation occurs when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script which can handle all the C programs (i.e. those file ending in ".c"). Thus the following fragment will optimize the executions of make for maintaining and archive library:

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)c.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        ar rv $(LIB) $?
        rm $?
```

Here, finally, is a legitimate use for the double colon. A dependency of the preceding form would be necessary for each of the different types of source files (suffices) which define the archive library. These translations are added in an effort to make more general use of the wealth of information which **make** generates.

3.12 The Test makefile

A test *makefile* was written to explicitly test each of the new features. When shipped to a new machine and compiled, **make** is assumed to work if the test *makefile* executes without error.

4. INCOMPATIBILITIES WITH OLD VERSION

The only known incompatibility with UNIX/TS **make** is seen in the following example makefile:

```
all:    cat dd
```

```
dd:     dd.o
        $(CC) -o $@ $?

cat:    cat.o
        $(CC) -o $@ $?
```

UNIX/TS **make** will not complain that *all* does not have a rule associated with it. The current version will. The current version is a strict interpretation of the original paper by S. I. Feldman and as such is described by his paper. The UNIX/TS **make** is wrong (according to Feldman's paper) but people have learned (through trial and error) to use it in this fashion and would resist the change. The differences amount to one line of code in the file *doname.c* which is noted in my source code. Furthermore, the "-b" option tells **make** to revert to the old method, whereby old makefiles can be run with this new version of **make**. Any other differences are unintentional.

5. FUTURE DEVELOPMENTS

Further developments of the **make** program that have been considered include redefinition of the colon, some other type of comparison that "newer versus older", searching out include files, and looking for source files in "other" directories. I will try to address each of these features.

Redefinition of the colon would effectively give the user the ability to provide **make** with information from a file, other than the time. Within the current syntax of **make** this does not seem too difficult:

```
make :=program ...
```

However, I can not see (nor has anyone pointed out to me) a use for such a feature. This would apparently be related to the second item above; namely, a different comparison other than the difference between two long integers, (the comparison by time). The basic problem, as I see it, is that when it comes time to do the ":" thing, **make** does not know what information to pass to the program. Make looks at each dependency individually and not in subsets. This prevents passing information like:

```
program cat cat.c
```

to a user defined *colon* routine because **make** does not know both *cat* and *cat.c* at the same time. (There is an interesting "hidden" variable in this version of **make**: "$!". It represents the current predecessor tree. In the following *makefile*:

```
all:    cat
        @echo cat up-to-date

cat:    cat.c
        echo $!
```

when the "echo $!" is executed, "$!" evaluates to

```
cat.c cat all
```

which is not *all* that useful! Further, it occasionally prints a message

```
$! nulled, predecessor circle
```

This message means that the predecessors of a file are circular. The actual evaluation of the "$!"

macro was aborted, and its value set to null. Otherwise there is no effect.)

The searching out of include files has been mentioned many times as an improvement. This would require **make** to look through every line of every source file mentioned in the makefile every time it is executed. This would slow **make** down to a slow crawl and slowly defeat its usefulness.

Having **make** look in other directories for file entries sounds useful. However, interesting problems arise when this is considered. What if the file in a remote directory is out of date with respect to a file in the current directory? Does **make** rebuild the remote file? (The user may not have write permission in that directory.) Also, how are the shell commands parameterized to be able to "see" the remote files. Further, how does **make** (or the programmer) guarantee the resulting target file is in the remote directory? (CC leaves the object in the current directory, which would mean that part of a command line might have to refer to a remote file while the rest of it would refer to a relative in the current directory.) I do not deny the usefulness of using remote directories, but cannot see a consistent solution to locating results. (I would be glad to here from anyone who can clear up the mess.)

6. CONCLUSION

The development described herein, produced a version of **make** which now serves the Columbus operating system group for maintenance of all of the UNIX source files. The *makefiles* supplied from the UNIX/TS support group are edited slightly (for the annoying incompatibility described above) and used intact. When a large improvement can be made, they are rewritten. When no makefile exists, (C library and most of the "cmd" directory) a *makefile* is written. There are no "*.rc" files left in the source. This gives a single interface to rebuilding parts or all of the UNIX software.

I feel that although the size of **make** has grown from

16320+3772+6352 = 26444b = 063514b

on the Center 127 machine to

21632+4850+8748 = 35230b = 0104636b

on the Columbus machine, the trade of size for functionality is worthwhile. The unfortunate by product of such a development is that there are two versions of the **make** program. I leave it to the reader to decide on the merits or lack of same on this issue.

CB-5255-EGB-egb **E. G. Bradford**

Atts.
Appendix A-Internal Definitions
Appendix B-Example Library Makefile
Appendix C-Example Recursive Use of Makefiles

APPENDIX A

The following *makefile* will exactly reproduce the internal rules of the current version of make. Thus if **make -r ...** is typed and a *makefile* includes this *makefile* the results would be identical to excluding the "-r" option and the *include* line in the makefile.

```
#       LIST OF SUFFIXES
.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .sh .sh~ .h .h~

#       PRESET VARIABLES
MAKE=make
YACC=yacc
YFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
CC=cc
CFLAGS=-O
AS=as
ASFLAGS=
GET=get
GFLAGS=

#       SINGLE SUFFIX RULES
.c:
        $(CC) -n -O $< -o $@
.c~:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) -n -O $*.c -o $*
        -rm -f $*.c
.sh:
        cp $< $@
.sh~:
        $(GET) $(GFLAGS) -p $< > $*.sh
        cp $*.sh $*
        -rm -f $*.sh

#       DOUBLE SUFFIX RULES
.c.o:
        $(CC) $(CFLAGS) -c $<
.c~.o:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
.c~.c:
        $(GET) $(GFLAGS) -p $< > $*.c
.s.o:
        $(AS) $(ASFLAGS) -o $@ $<
.s~.o:
        $(GET) $(GFLAGS) -p $< > $*.s
        $(AS) $(ASFLAGS) -o $*.o $*.s
        -rm -f $*.s
.y.o:
```

```
	$(YACC) $(YFLAGS) $<
	$(CC) $(CFLAGS) -c y.tab.c
	rm y.tab.c
	mv y.tab.o $@
.y~.o:
	$(GET) $(GFLAGS) -p $< > $*.y
	$(YACC) $(YFLAGS) $*.y
	$(CC) $(CFLAGS) -c y.tab.c
	rm -f y.tab.c $*.y
	mv y.tab.o $*.o
.l.o:
	$(LEX) $(LFLAGS) $<
	$(CC) $(CFLAGS) -c lex.yy.c
	rm lex.yy.c
	mv lex.yy.o $@
.l~.o:
	$(GET) $(GFLAGS) -p $< > $*.l
	$(LEX) $(LFLAGS) $*.l
	$(CC) $(CFLAGS) -c lex.yy.c
	rm -f lex.yy.c $*.l
	mv lex.yy.o $*.o
.y.c :
	$(YACC) $(YFLAGS) $<
	mv y.tab.c $@
.y~.c :
	$(GET) $(GFLAGS) -p $< > $*.y
	$(YACC) $(YFLAGS) $*.y
	mv y.tab.c $*.c
	-rm -f $*.y
.l.c :
	$(LEX) $<
	mv lex.yy.c $@
.c.a:
	$(CC) -c $(CFLAGS) $<
	ar rv $@ $*.o
	rm -f $*.o
.c~.a:
	$(GET) $(GFLAGS) -p $< > $*.c
	$(CC) -c $(CFLAGS) $*.c
	ar rv $@ $*.o
	rm -f $*.[co]
.s~.a:
	$(GET) $(GFLAGS) -p $< > $*.s
	$(AS) $(ASFLAGS) -o $*.o $*.s
	ar rv $@ $*.o
	-rm -f $*.[so]
.h~.h:
	$(GET) $(GFLAGS) -p $< > $*.h
```

APPENDIX B

The following library maintaining makefile is from my current work on LSX. It completely maintains the LSX operating system library.

```
#       @(#)/usr/src/cmd/make/make.tm       3.2
LIB = lsxlib

#       I have a banner printing pr.
PR = vpr -b LSX

INSDIR = /r1/flop0/
INS = eval

lsx::   $(LIB) low.o mch.o
        ld -x low.o mch.o $(LIB)
        mv a.out lsx
        @size lsx

#       Here, I have used $(INS) as either ':' or 'eval'.
lsx::
        $(INS) 'cp lsx $(INSDIR)lsx && \
                strip $(INSDIR)lsx  && \
                ls -l $(INSDIR)lsx'

print:
        $(PR) header.s low.s mch.s *.h *.c Makefile

$(LIB): \
        $(LIB)(clock.o) \
        $(LIB)(main.o) \
        $(LIB)(tty.o) \
        $(LIB)(trap.o) \
        $(LIB)(sysent.o) \
        $(LIB)(sys2.o) \
        $(LIB)(sys3.o) \
        $(LIB)(sys4.o) \
        $(LIB)(sys1.o) \
        $(LIB)(sig.o) \
        $(LIB)(fio.o) \
        $(LIB)(k1.o) \
        $(LIB)(alloc.o) \
        $(LIB)(nami.o) \
        $(LIB)(iget.o) \
        $(LIB)(rdwri.o) \
        $(LIB)(subr.o) \
        $(LIB)(bio.o) \
        $(LIB)(decfd.o) \
        $(LIB)(slp.o) \
        $(LIB)(space.o) \
        $(LIB)(puts.o)
        @echo $(LIB) now up-to-date.
```

```
.s.o:
        as -o $*.o header.s $*.s
.o.a:
        ar rv $@ $<
        rm -f $<

.s.a:
        as -o $*.o header.s $*.s
        ar rv $@ $*.o
        rm -f $*.o

.PRECIOUS:      $(LIB)
```

APPENDIX C

The following set of *makefiles* maintain the UNIX operating system for Columbus UNIX. They reside in the following relative directories on the Columbus operating systems group machine: "ucb", "ucb/os", "ucb/io", "ucb/head/sys". Each one is named "70.mk". The following command forces a complete rebuild of the operating system:

```
FRC=FRC make -f 70.mk
```

where the current directory is "ucb". Here, I have used some of the conventions described in A. Chellis' paper, *Proposed Structure for UNIX/TS and UNIX/RT Makefiles* (MF78-8234-73). FRC is a convention for *F*o*R*C*C*ing **make** to completely rebuild a target starting from scratch.

```
#       @(#)/usr/src/cmd/make/make.tm       3.2
#       ucb/70.mk makefile

VERSION = 70

DEPS = \
        os/low.$(VERSION).o \
        os/mch.$(VERSION).o \
        os/conf.$(VERSION).o \
        os/lib1.$(VERSION).a \
        io/lib2.$(VERSION).a

#       This makefile will re-load unix.$(VERSION) if any
#       of the $(DEPS) is out-of-date wrt unix.$(VERSION).
#       Note, it will not go out and check each member
#       of the libraries. To do this, the FRC macro must
#       be defined.

unix.$(VERSION):        $(DEPS) $(FRC)
        load -s $(VERSION)

$(DEPS):        $(FRC)
        cd $(@D); $(MAKE) -f $(VERSION).mk $(@F)

all:    unix.$(VERSION)
        @echo unix.$(VERSION) up-to-date.

includes:
        cd head/sys; $(MAKE) -f $(VERSION).mk

FRC:    includes;
```

```
#       @(#)/usr/src/cmd/make/make.tm       3.2
#       ucb/os/70.mk makefile

VERSION = 70

LIB = lib1.$(VERSION).a
COMPOOL=

LIBOBJS = \
        $(LIB)(main.o) \
        $(LIB)(alloc.o) \
        $(LIB)(iget.o) \
        $(LIB)(prf.o) \
        $(LIB)(rdwri.o) \
        $(LIB)(slp.o) \
        $(LIB)(subr.o) \
        $(LIB)(text.o) \
        $(LIB)(trap.o) \
        $(LIB)(sig.o) \
        $(LIB)(sysent.o) \
        $(LIB)(sys1.o) \
        $(LIB)(sys2.o) \
        $(LIB)(sys3.o) \
        $(LIB)(sys4.o) \
        $(LIB)(sys5.o) \
        $(LIB)(syscb.o) \
        $(LIB)(maus.o) \
        $(LIB)(messag.o) \
        $(LIB)(nami.o) \
        $(LIB)(fio.o) \
        $(LIB)(clock.o) \
        $(LIB)(acct.o) \
        $(LIB)(errlog.o)


ALL = \
        conf.$(VERSION).o \
        low.$(VERSION).o \
        mch.$(VERSION).o \
        $(LIB)

all:    $(ALL)
        @echo '$(ALL)' now up-to-date.

$(LIB):: $(LIBOBJS)

$(LIBOBJS):     $(FRC)

FRC:
        rm -f $(LIB)

clobber:cleanup
```

```
	-rm -f $(LIB)

clean cleanup:

install: all

.PRECIOUS:	$(LIB)
```

```
#       @(#)/usr/src/cmd/make/make.tm       3.2
#       ucb/io/70.mk makefile

VERSION = 70

LIB = lib2.$(VERSION).a
COMPOOL=

LIB2OBJS = \
        $(LIB)(mx1.o) \
        $(LIB)(mx2.o) \
        $(LIB)(bio.o) \
        $(LIB)(tty.o) \
        $(LIB)(malloc.o) \
        $(LIB)(pipe.o) \
        $(LIB)(dhdm.o) \
        $(LIB)(dh.o) \
        $(LIB)(dhfdm.o) \
        $(LIB)(dj.o) \
        $(LIB)(dn.o) \
        $(LIB)(ds40.o) \
        $(LIB)(dz.o) \
        $(LIB)(alarm.o) \
        $(LIB)(hf.o) \
        $(LIB)(hps.o) \
        $(LIB)(hpmap.o) \
        $(LIB)(hp45.o) \
        $(LIB)(hs.o) \
        $(LIB)(ht.o) \
        $(LIB)(jy.o) \
        $(LIB)(kl.o) \
        $(LIB)(lfh.o) \
        $(LIB)(lp.o) \
        $(LIB)(mem.o) \
        $(LIB)(nmpipe.o) \
        $(LIB)(rf.o) \
        $(LIB)(rk.o) \
        $(LIB)(rp.o) \
        $(LIB)(rx.o) \
        $(LIB)(sys.o) \
        $(LIB)(trans.o) \
        $(LIB)(ttdma.o) \
        $(LIB)(tec.o) \
        $(LIB)(tex.o) \
        $(LIB)(tm.o) \
        $(LIB)(vp.o) \
        $(LIB)(vs.o) \
        $(LIB)(vtlp.o) \
        $(LIB)(vt11.o) \
        $(LIB)(fakevtlp.o) \
        $(LIB)(vt61.o) \
        $(LIB)(vt100.o) \
```

```
        $(LIB)(vtmon.o) \
        $(LIB)(vtdbg.o) \
        $(LIB)(vtutil.o) \
        $(LIB)(vtast.o) \
        $(LIB)(partab.o) \
        $(LIB)(rh.o) \
        $(LIB)(devstart.o) \
        $(LIB)(dmc11.o) \
        $(LIB)(rop.o) \
        $(LIB)(ioctl.o) \
        $(LIB)(fakemx.o)

all:    $(LIB)
        @echo $(LIB) is now up-to-date.

$(LIB):: $(LIB2OBJS)

$(LIB2OBJS):    $(FRC)

FRC:
        rm -f $(LIB)

clobber: cleanup
        -rm -f $(LIB) *.o

clean cleanup:

install: all

.PRECIOUS:      $(LIB)

.s.a:
        $(AS) $(ASFLAGS) -o $*.o $<
        ar rcv $@ $*.o
        rm $*.o
```

```
#       @(#)/usr/src/cmd/make/make.tm       3.2
#       ucb/head/sys/70.mk makefile

COMPOOL = /usr/include/sys

HEADERS = \
        $(COMPOOL)/buf.h \
        $(COMPOOL)/bufx.h \
        $(COMPOOL)/conf.h \
        $(COMPOOL)/confx.h \
        $(COMPOOL)/crtctl.h \
        $(COMPOOL)/dir.h \
        $(COMPOOL)/dm11.h \
        $(COMPOOL)/elog.h \
        $(COMPOOL)/file.h \
        $(COMPOOL)/filex.h \
        $(COMPOOL)/filsys.h \
        $(COMPOOL)/ino.h \
        $(COMPOOL)/inode.h \
        $(COMPOOL)/inodex.h \
        $(COMPOOL)/ioctl.h \
        $(COMPOOL)/ipcomm.h \
        $(COMPOOL)/ipcommx.h \
        $(COMPOOL)/lfsh.h \
        $(COMPOOL)/lock.h \
        $(COMPOOL)/maus.h \
        $(COMPOOL)/mx.h \
        $(COMPOOL)/param.h \
        $(COMPOOL)/proc.h \
        $(COMPOOL)/procx.h \
        $(COMPOOL)/reg.h \
        $(COMPOOL)/seg.h \
        $(COMPOOL)/sgtty.h \
        $(COMPOOL)/sigdef.h \
        $(COMPOOL)/sprof.h \
        $(COMPOOL)/sprofx.h \
        $(COMPOOL)/stat.h \
        $(COMPOOL)/syserr.h \
        $(COMPOOL)/sysmes.h \
        $(COMPOOL)/sysmesx.h \
        $(COMPOOL)/systm.h \
        $(COMPOOL)/text.h \
        $(COMPOOL)/textx.h \
        $(COMPOOL)/timeb.h \
        $(COMPOOL)/trans.h \
        $(COMPOOL)/tty.h \
        $(COMPOOL)/ttyx.h \
        $(COMPOOL)/types.h \
        $(COMPOOL)/user.h \
        $(COMPOOL)/userx.h \
        $(COMPOOL)/version.h \
        $(COMPOOL)/votrax.h \
```

```
        $(COMPOOL)/vt11.h \
        $(COMPOOL)/vtmn.h

all:    $(FRC) $(HEADERS)
        @echo Headers are now up to date.

$(HEADERS): s.$$/
        $(GET) -s -p $(GFLAGS) $? > xtemp
        move xtemp 444 src sys $@

FRC:
        rm -f $(HEADERS)

.PRECIOUS:    $(HEADERS)


.h~.h:
        get -s $<

.DEFAULT:
        cpmv $? 444 src sys $@
```

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 doesn't 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 require 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 less than or equal to 9999, which may be retrieved by a *get*(1) command for editing. The default value for an unspecified c flag is 9999.

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

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

i 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.

j Allows concurrent *get*(1) commands for editing on the same SID of an SCCS file. This allows multiple concurrent updates to the same version of the SCCS file.

l*list* A *list* of releases to which deltas can no longer be made (get —e against one of these "locked" releases fails). The *list* has the following syntax:

<list> ::= <range> | <list> , <range>
<range> ::= *RELEASE NUMBER* | a

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

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

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

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

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

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

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

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

—**a***login* A *login* name, or numerical UNIX System group ID, to be added to the list of users which may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all *login* names common to that group ID. Several **a** keyletters may be used on a single *admin* command line. As many *logins*, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas.

—**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.

—**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).

—**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.

—**h** Causes *admin* to check the structure of the SCCS file (see *sccsfile*(5)), and to compare a newly computed check-sum (the sum of all the characters in the SCCS file except those in the first line) with the check-sum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced.

This keyletter inhibits writing on the file, so that it nullifies the effect of any other keyletters supplied, and is, therefore, only meaningful when processing existing files.

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

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

FILES

The last component of all SCCS file names must be of the form **s.***file-name*. New SCCS files are given mode 444 (see *chmod*(1)). Write permission in the pertinent directory is, of course, required to create a file. All writing done by *admin* is to a temporary x-file, called **x.***file-name*, (see *get*(1)), created with mode 444 if the *admin* command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of *admin*, the SCCS file is removed (if it exists), and the x-file is renamed with the name of the SCCS file. This ensures that changes are made to the SCCS file only if no errors occurred.

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

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

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

SEE ALSO

delta(1), ed(1), get(1), help(1), prs(1), what(1), sccsfile(4).
Source Code Control System User's Guide in the *UNIX System User's Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

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* Specifies 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 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

All files of the form ?-file are explained in the *Source Code Control System User's Guide*. The naming convention for these files is also described there.

g-file	Existed before the execution of *delta*; removed after completion of *delta*.
p-file	Existed before the execution of *delta*; may exist after completion of *delta*.
q-file	Created during the execution of *delta*; removed after completion of *delta*.
x-file	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z-file	Created during the execution of *delta*; removed during the execution of *delta*.
d-file	Created during the execution of *delta*; removed after completion of *delta*.
/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*(5)) and will cause an error.

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

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

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

SEE ALSO

admin(1), bdiff(1), cdc(1), get(1), help(1), prs(1), rmdel(1), sccsfile(4).
Source Code Control System User's Guide in the *UNIX System User's Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

get — get a version of an SCCS file

SYNOPSIS

get [—rSID] [—ccutoff] [—ilist] [—xlist] [—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

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

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

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

—**b** Used with the —**e** keyletter to indicate that the new delta should have an SID in a new branch as shown in Table 1. This keyletter is ignored if the **b** flag is not

present in the file (see *admin*(1)) or if the retrieved *delta* is not a leaf *delta*. (A leaf *delta* is one that has no successors on the SCCS file tree.)
Note: A branch *delta* may always be created from a non-leaf *delta*.

—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. Partial SIDs are interpreted as shown in the "SID Retrieved" column of Table 1.

—x*list* A *list* of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the —**i** keyletter for the *list* format.

—**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 ("top") delta in a given release (e.g., —**r1**), or release and level (e.g., —**r1.2**).

—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, and users should not use it. If both the —**r** and —**a** keyletters are specified, 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 shouldn't 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 the UNIX System program files. %W% = %Z%%M%<horizontal-tab>%I%
%A%	Another shorthand notation for constructing *what*(1) strings for non-UNIX System program files. %A% = %Z%%Y% %M% %I%%Z%

FILES

Several auxiliary files may be created by *get*, These files are known generically as the *g-file*, *l-file*, *p-file*, and *z-file*. The letter before the hyphen is called the tag. An auxiliary 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 wasn't applied and ignored;
* if the delta wasn't applied and wasn't 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 newline. 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.

SEE ALSO

admin(1), delta(1), help(1), prs(1), what(1), sccsfile(4).
Source Code Control System in the *UNIX System Support Tools Guide*.

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 doesn't, then only one file may be named when the —e keyletter is used.

NAME

sccsdiff — compare two versions of an SCCS file

SYNOPSIS

sccsdiff —rSID1 —rSID2 [—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

bdiff(1), get(1), help(1), pr(1).
Source Code Control System UNIX System User's Guide.

DIAGNOSTICS

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

NAME

sccsfile - format of SCCS file

DESCRIPTION

An SCCS file is an ASCII file. It consists of six logical parts: the *checksum*, the *delta table* (contains information about each delta), *user names* (contains login names and/or numerical group IDs of users who may add deltas), *flags* (contains definitions of internal keywords), *comments* (contains arbitrary descriptive information about the file), and the *body* (contains actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the **ASCII SOH** (start of heading) character (octal 001). This character is hereafter referred to as *the control character* and will be represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form **DDDDD** represent a five digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

@hDDDDD

The value of the checksum is the sum of all characters, except those of the first line. The @h provides a *magic number* of (octal) 064001.

Delta table

The delta table consists of a variable number of entries of the form:

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> DDDDD DDDDD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
 .
 .
 .
@c <comments> ...
 .
 .
 .
@e
```

The first line (@s) contains the number of lines inserted/deleted/unchanged respectively. The second line (@d) contains the type of the delta (currently, normal: **D**, and removed: **R**), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The @i, @x, and @g lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The @m lines (optional) each contain one MR number associated with the delta; the

...es contain comments associated with the delta.

...e line ends the delta table entry.

User ...

...t of login names and/or numerical group IDs of users who may add deltas to ...e, separated by new-lines. The lines containing these login names and/or ...ical group IDs are surrounded by the bracketing lines **@u** and **@U**. An empty ...lows anyone to make a delta.

Flags

...rds used internally (see *admin*(1) for more information on their use). Each ...e takes the form:

@f <flag> <optional text>

...following flags are defined:

@f t <type of program>
@f v <program name>
@f i
@f b
@f m <module name>
@f f <floor>
@f c <ceiling>
@f d <default-sid>
@f n
@f j
@f l <lock-releases>
@f q <user defined>
@f z <reserved for use in interfaces>

...flag defines the replacement for the **%Y%** identification keyword. The **v** flag ...s prompting for MR numbers in addition to comments; if the optional text is ...t it defines an MR number validity checking program. The **i** flag controls the ...ng/error aspect of the "No id keywords" message. When the **i** flag is not ...t, this message is only a warning; when the **i** flag is present, this message will ...a "fatal" error (the file will not be gotten, or the delta will not be made). ... the **b** flag is present the **—b** keyletter may be used on the *get* command to ... a branch in the delta tree. The **m** flag defines the first choice for the replace- ...text of the %M% identification keyword. The **f** flag defines the "floor" release; ...lease below which no deltas may be added. The **c** flag defines the "ceiling" ...e; the release above which no deltas may be added. The **d** flag defines the ...lt SID to be used when none is specified on a *get* command. The **n** flag causes ...to insert a "null" delta (a delta that applies *no* changes) in those releases that ...ipped when a delta is made in a *new* release (e.g., when delta 5.1 is made after ...2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped ...es to be completely empty. The **j** flag causes *get* to allow concurrent edits of ...me base SID. The **l** flag defines a *list* of releases that are *locked* against editing ...(1) with the **—e** keyletter). The **q** flag defines the replacement for the **%Q%** ...tification keyword. **z** flag is used in certain specialized interface programs.

Comments

...rary text surrounded by the bracketing lines **@t** and **@T**. The comments sec- ...n typically will contain a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines don't begin with the control character, control lines do. There are three kinds of control lines: *insert*, *delete*, and *end*, represented by:

@I DDDDD
@D DDDDD
@E DDDDD

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

admin(1), delta(1), get(1), prs(1).
Source Code Control System User's Guide in the *UNIX System User's Guide*.

pic - Tgms

Bell Laboratories
Murray Hill, New Jersey 07974

Computing Science Technical Report No. 85

PIC — A Graphics Language for Typesetting User Manual

Brian W. Kernighan

Revised Edition, March, 1982

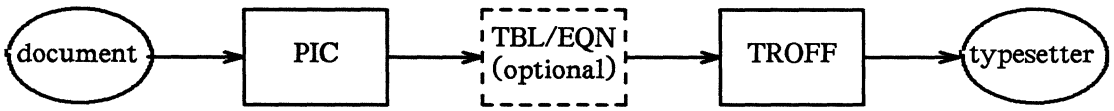
PIC — A Graphics Language for Typesetting
User Manual

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

PIC is a language for drawing simple figures on a typesetter. The basic objects in PIC are boxes, circles, ellipses, lines, arrows, arcs, spline curves, and text. These may be placed anywhere, at positions specified absolutely or in terms of previous objects. The example below illustrates the general capabilities of the language.



This picture was created with the input

```
ellipse "document"
arrow
box "PIC"
arrow
box "TBL/EQN" "(optional)" dashed
arrow
box "TROFF"
arrow
ellipse "typesetter"
```

PIC is another TROFF processor; it passes most of its input through untouched, but translates commands between *.PS* and *.PE* into TROFF commands that draw the pictures.

Revised Edition, March, 1982

PIC — A Graphics Language for Typesetting
User Manual

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

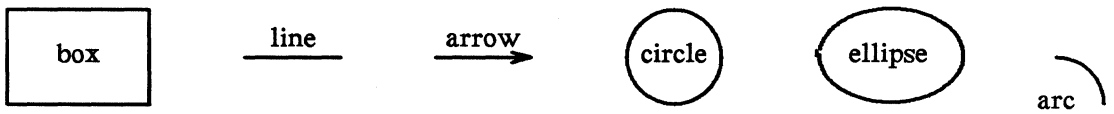
PIC is a language for drawing simple pictures. It operates as yet another TROFF[1] preprocessor, (in the same style as EQN[2], TBL[3] and REFER[4]), with pictures marked by *.PS* and *.PE*.

PIC was inspired partly by Chris Van Wyk's early work on IDEAL[5]; it has somewhat the same capabilities, but quite a different flavor. In particular, PIC is much more procedural—a picture is drawn by specifying (sometimes in painful detail) the motions that one goes through to draw it. Other direct influences include the PICTURE language [6] and the V viewgraph language [7].

This paper is primarily a user's manual for PIC; a discussion of design issues and user experience may be found in [8]. The next section shows how to use PIC in the most simple way. Subsequent sections describe how to change the sizes of objects when the defaults are wrong, and how to change their positions when the standard positioning rules are wrong. An appendix describes the language succinctly and more or less precisely.

2. Basics

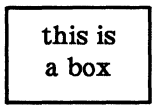
PIC provides boxes, lines, arrows, circles, ellipses, arcs, and splines (arbitrary smooth curves), plus facilities for positioning and labeling them. The picture below shows all of the fundamental objects (except for splines) in their default sizes:



Each picture begins with *.PS* and ends with *.PE*; between them are commands to describe the picture. Each command is typed on a line by itself. For example

```
.PS
box "this is" "a box"
.PE
```

creates a standard box (¾ inch wide, ½ inch high) and centers the two pieces of text in it:



Each line of text is a separate quoted string. Quotes are mandatory, even if the text contains no blanks. (Of course there needn't be any text at all.) Each line will be printed in the current size and font, centered horizontally, and separated vertically by the current TROFF line spacing.

PIC does not center the drawing itself, but the default definitions of *.PS* and *.PE* in the

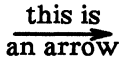
-ms macro package do.

You can use *circle* or *ellipse* in place of *box*:

Text is centered on lines and arrows; if there is more than one line of text, the lines are centered above and below:

```
.PS
arrow "this is" "an arrow"
.PE
```

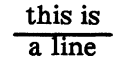
produces



and

```
line "this is" "a line"
```

gives



Boxes and lines may be dashed or dotted; just add the word *dashed* or *dotted* after *box* or *line*.

Arcs by default turn 90 degrees counterclockwise from the current direction; you can make them turn clockwise by saying *arc cw*. So

```
line; arc; arc cw; arrow
```

produces

A spline might well do this job better; we will return to that shortly.

As you might guess,

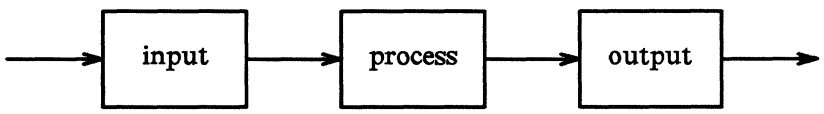
```
arc; arc; arc; arc
```

draws a circle, though not very efficiently.

Objects are normally drawn one after another, left to right, and connected at the obvious places. Thus the input

```
arrow; box "input"; arrow; box "process"; arrow; box "output"; arrow
```

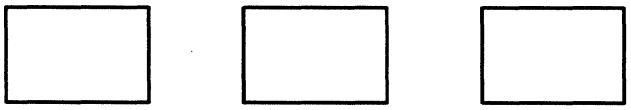
produces the figure



If you want to leave a space at some place, use *move*:

```
box; move; box; move; box
```

produces

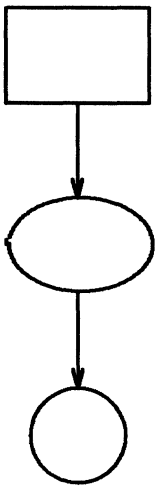


Notice that several commands can be put on a single line if they are separated by semicolons.

Although objects are normally connected left to right, this can be changed. If you specify a direction (as a separate object), subsequent objects will be joined in that direction. Thus

```
down; box; arrow; ellipse; arrow; circle
```

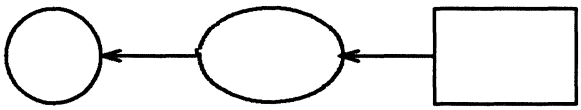
produces



and

```
left; box; arrow; ellipse; arrow; circle
```

produces



Each new picture begins going to the right.

Normally, figures are drawn at a fixed scale, with objects of a standard size. It is possible, however, to arrange that a figure be expanded to fit a particular width. If the *.PS* line contains a number, the drawing is forced to be that many inches wide, with the height scaled proportionately. Thus

```
.PS 3.5i
```

causes the picture to be 3.5 inches wide.

PIC is pretty dumb about the size of text in relation to the size of boxes, circles, and so on. There is as yet no way to say "make a box that just fits around this text" or "make this text fit inside this circle" or "draw a line as long as this text." All of these facilities are useful, so the limitations may go away in the fullness of time, but don't hold your breath. In the meantime, tight fitting of text can generally only be done by trial and error.

Speaking of errors, if you make a grammatical error in the way you describe a picture, PIC will complain and try to indicate where. For example, the invalid input

```
box arrow box
```

will draw the message

```
pic: syntax error near line 5, file -
 context is
      box arrow ^ box
```

The caret ^ marks the place where the error was first noted; it typically *follows* the word in error.

3. Controlling Sizes

This section deals with how to control the sizes of objects when the "default" sizes are not what is wanted. The next section deals with positioning them when the default positions are not right.

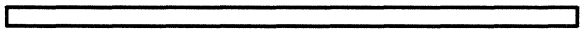
Each object that PIC knows about (boxes, circles, etc.) has associated dimensions, like height, width, radius, and so on. By default, PIC tries to choose sensible default values for these dimensions, so that simple pictures can be drawn with a minimum of fuss and bother. All of the figures and motions shown so far have been in their default sizes:

box	¾" wide × ½" high
circle	½" diameter
ellipse	¾" wide × ½" high
arc	½" radius
line or arrow	½" long
move	½" in the current direction

When necessary, you can make any object any size you want. For example, the input

```
box width 3i height 0.1i
```

draws a long, flat box



3 inches wide and 1/10 inch high. There must be no space between the number and the "*i*" that indicates a measurement in inches. In fact, the "*i*" is optional; all positions and dimensions are taken to be in inches.

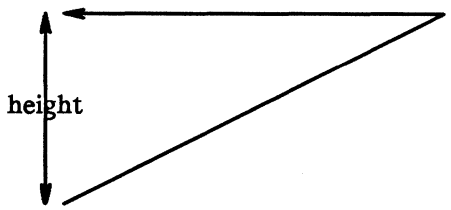
Giving an attribute like *width* changes only the one instance of the object. You can also change the default size for all objects of a particular type, as discussed later.

The attributes of *height* (which you can abbreviate to *ht*) and *width* (or *wid*) apply to boxes, circles, ellipses, and to the head on an arrow. The attributes of *radius* (or *rad*) and *diameter* (or *diam*) can be used for circles and arcs if they seem more natural.

Lines and arrows are most easily drawn by specifying the amount of motion from where one is right now, in terms of directions. Accordingly the words *up*, *down*, *left* and *right* and an optional distance can be attached to *line*, *arrow*, and *move*. For example,

```
.PS
line up 1i right 2i
arrow left 2i
move left 0.1i
line <-> down 1i "height"
.PE
```

draws

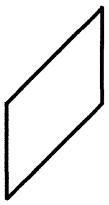


The notation <-> indicates a two-headed arrow; use -> for a head on the end and <- for one on the start. Lines and arrows are really the same thing; in fact, *arrow* is a synonym for *line* ->.

If you don't put any distance after *up*, *down*, etc., PIC uses the standard distance. So

```
line up right; line down; line down left; line up
```

draws the parallelogram



Warning: a very common error (which hints at a language defect) is to say

```
line 3i
```

A direction is needed:

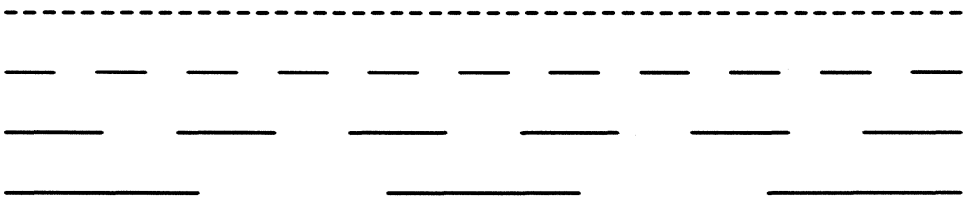
```
line right 3i
```

Boxes and lines may be dotted or dashed:

comes from

```
box dotted; line dotted; move; line dashed
```

If there is a number after *dot*, the dots will be that far apart. You can also control the size of the dashes (at least somewhat): if there is a length after the word *dashed*, the dashes will be that long, and the intervening spaces will be as close as possible to that size. So, for instance,



comes from the inputs (as separate pictures)

```
line right 5i dashed
line right 5i dashed 0.25i
line right 5i dashed 0.5i
line right 5i dashed 1i
```

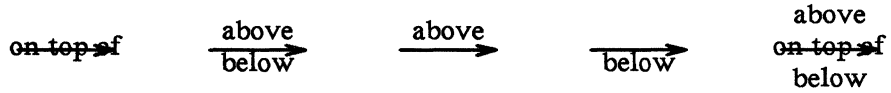
Sorry, but circles and arcs can't be dotted or dashed yet, and probably never will be.

You can make any object invisible by adding the word *invis(ible)* after it. This is particularly useful for positioning things correctly near text, as we will see later.

Text may be positioned on lines and arrows:

```
.PS
arrow "on top of"; move
arrow "above" "below"; move
arrow "above" above; move
arrow "below" below; move
arrow "above" "on top of" "below"
.PE
```

produces



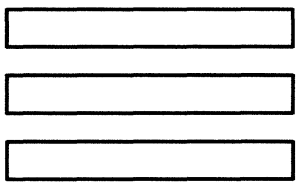
The "width" of an arrowhead is the distance across its tail; the "height" is the distance along the shaft. The arrowheads in this picture are default size.

As we said earlier, arcs go 90 degrees counterclockwise from where you are right now, and *arc cw* changes this to clockwise. The default radius is the same as for circles, but you can change it with the *rad* attribute. It is also easy to draw arcs between specific places; this will be described in the next section.

To put an arrowhead on an arc, use one of <-, -> or <->.

In all cases, unless an explicit dimension for some object is specified, you will get the default size. If you want an object to have the same size as the previous one of that kind, add the word *same*. Thus in the set of boxes given by

```
down; box ht 0.2i wid 1.5i; move down 0.15i; box same; move same; box same
```



the dimensions set by the first *box* are used several times; similarly, the amount of motion for the second *move* is the same as for the first one.

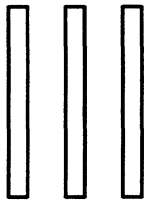
It is possible to change the default sizes of objects by assigning values to certain variables:

```
boxwid, boxht
linewid, lineht
dashwid
circlerad
arcrad
ellipsewid, ellipseht
movewid, moveht
arrowwid, arrowht       (These refer to the arrowhead.)
```

So if you want all your boxes to be long and skinny, and relatively close together,

```
boxwid = 0.1i; boxht = 1i
movewid = 0.2i
box; move; box; move; box
```

gives



PIC works internally in what it thinks are inches. Setting the variable *scale* to some value causes all dimensions to be scaled down by that value. Thus, for example, *scale=2.54* causes dimensions to be interpreted as centimeters.

The number given as a width in the *.PS* line overrides the dimensions given in the picture; this can be used to force a picture to a particular size even when coordinates have been given in inches. Experience indicates that the easiest way to get a picture of the right size is to enter its dimensions in inches, then if necessary add a width to the *.PS* line.

4. Controlling Positions

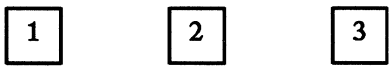
You can place things anywhere you want; PIC provides a variety of ways to talk about places. PIC uses a standard Cartesian coordinate system, so any point or object has an x and y position. The first object is placed with its start at position 0,0 by default. The x,y position of a box, circle or ellipse is its geometrical center; the position of a line or motion is its beginning; the position of an arc is the center of the corresponding circle.

Position modifiers like *from*, *to*, *by* and *at* are followed by an x,y pair, and can be attached to boxes, circles, lines, motions, and so on, to specify or modify a position.

You can also use *up*, *down*, *right*, and *left* with *line* and *move*. Thus

```
.PS 2
box ht 0.2 wid 0.2 at 0,0 "1"
move to 0.5,0           # or "move right 0.5"
box "2" same            # use same dimensions as last box
move same               # use same motion as before
box "3" same
.PE
```

draws three boxes, like this:



Note the use of *same* to repeat the previous dimensions instead of reverting to the default values.

Comments can be used in pictures; they begin with a # and end at the end of the line.

Attributes like *ht* and *wid* and positions like *at* can be written out in any order. So

```
box ht 0.2 wid 0.2 at 0,0
box at 0,0 wid 0.2 ht 0.2
box ht 0.2 at 0,0 wid 0.2
```

are all equivalent, though the last is harder to read and thus less desirable.

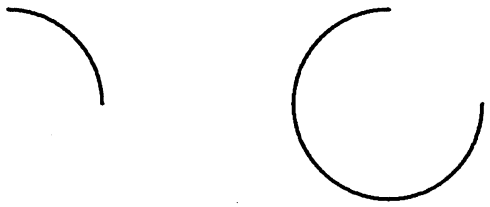
The *from* and *to* attributes are particularly useful with arcs, to specify the endpoints. By default, arcs are drawn counterclockwise,

```
arc from 0.5i,0 to 0,0.5i
```

is the short arc and

```
arc from 0,0.5i to 0.5i,0
```

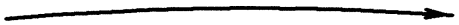
is the long one:



If the *from* attribute is omitted, the arc starts where you are now and goes to the point given by *to*. The radius can be made large to provide flat arcs:

```
arc -> cw from 0,0 to 2i,0 rad 15i
```

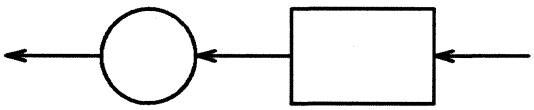
produces



We said earlier that objects are normally connected left to right. This is an oversimplification. The truth is that objects are connected together in the direction specified by the most recent *up*, *down*, *left* or *right* (either alone or as part of some object). Thus, in

```
arrow left; box; arrow; circle; arrow
```

the *left* implies connection towards the left:



This could also be written as

```
left; arrow; box; arrow; circle; arrow
```

Objects are joined in the order determined by the last *up*, *down*, etc., with the entry point of the second object attached to the exit point of the first. Entry and exit points for boxes, circles and ellipses are on opposite sides, and the start and end of lines, motions and arcs. It's not entirely clear that this automatic connection and direction selection is the right design, but it seems to simplify many examples.

If a set of commands is enclosed in braces {...}, the current position and direction of motion when the group is finished will be exactly where it was when entered. Nothing else is restored. There is also a more general way to group objects, using *[* and *]*, which is discussed in a later section.

5. Labels and Corners

Objects can be labelled or named so that you can talk about them later. For example,

```
.PS
Box1:
        box ...
        # ... other stuff ...
        move to Box1
.PE
```

Place names have to begin with an upper case letter (to distinguish them from variables, which begin with lower case letters). The name refers to the "center" of the object, which is the geometric center for most things. It's the beginning for lines and motions.

Other combinations also work:

```
line from Box1 to Box2
move to Box1 up 0.1 right 0.2
move to Box1 + 0.2,0.1  # same as previous
line to Box1 - 0.5,0
```

The reserved name *Here* may be used to record the current position of some object, for example as

```
Box1: Here
```

Labels are variables — they can be reset several times in a single picture, so a line of the form

```
Box1: Box1 + 1i,1i
```

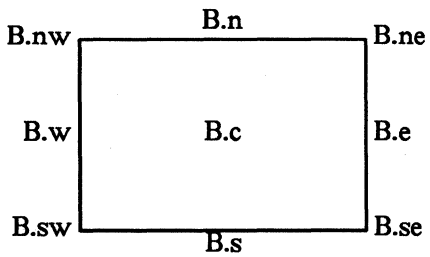
is perfectly legal.

You can also refer to previously drawn objects of each type, using the word *last*. For example, given the input

```
box "A"; circle "B"; box "C"
```

then *'last box'* refers to box *C*, *'last circle'* refers to circle *B*, and *'2nd last box'* refers to box *A*. Numbering of objects can also be done from the beginning, so boxes *A* and *C* are *'1st box'* and *'2nd box'* respectively.

To cut down the need for explicit coordinates, most objects have "corners" named by compass points:



The primary compass points may also be written as *.r*, *.b*, *.l*, and *.t*, for *right*, *bottom*, *left*, and *top*. The box above was produced with

```
.PS
B: box "B.c"
" B.e" at B.e ljust
" B.ne" at B.ne ljust
" B.se" at B.se ljust
"B.s" at B.s below
"B.n" at B.n above
"B.sw " at B.sw rjust
"B.w " at B.w rjust
"B.nw " at B.nw rjust
.PE
```

Note the use of *ljust*, *rjust*, *above*, and *below* to alter the default positioning of text, and of a blank with some strings to help space them away from a vertical line.

Lines and arrows have a *start*, an *end* and a center in addition to corners. (Arcs have only a center, a start, and an end.) There are a host of (i.e., too many) ways to talk about the corners of an object. Besides the compass points, almost any sensible combination of *left*, *right*, *top*, *bottom*, *upper* and *lower* will work. Furthermore, if you don't like the '.' notation, as in

```
last box.ne
```

you can instead say

```
upper right of last box
```

Prolixity like

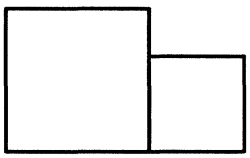
```
line from upper left of 2nd last box to bottom of 3rd last ellipse
```

begins to wear after a while, but it is descriptive. This part of the language is probably fat that will get trimmed.

It is sometimes easiest to position objects by positioning some part of one at some part of another, for example the northwest corner of one at the southeast corner of another. The *with* attribute in PIC permits this kind of positioning. For example,

```
box ht 0.75i wid 0.75i
box ht 0.5i wid 0.5i with .sw at last box.se
```

produces

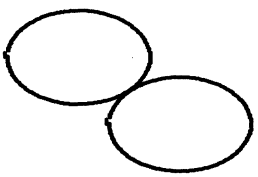


Notice that the corner after *with* is written *.sw*.

As another example, consider

```
ellipse; ellipse with .nw at last ellipse.se
```

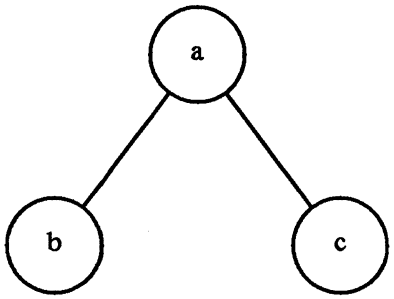
which makes



Sometimes it is desirable to have a line intersect a circle at a point which is not one of the eight compass points that PIC knows about. In such cases, the proper visual effect can be obtained by using the attribute *chop* to chop off part of the line:

```
circle "a"
circle "b" at 1st circle - (0.75i, 1i)
circle "c" at 1st circle + (0.75i, -1i)
line from 1st circle to 2nd circle chop
line from 1st circle to 3rd circle chop
```

produces



By default the line is chopped by *circlerad* at each end. This may be changed:

```
line ... chop r
```

chops both ends by *r*, and

```
line ... chop r1 chop r2
```

chops the beginning by *r1* and the end by *r2*.

There is one other form of positioning that is sometimes useful, to refer to a point some fraction of the way between two other points. This can be expressed in PIC as

fraction of the way between *position1* and *position2*

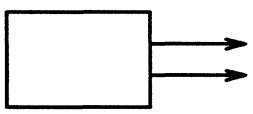
fraction is any expression, and *position1* and *position2* are any positions. You can abbreviate this rather windy phrase; "of the way" is optional, and the whole thing can be written instead as

fraction < *position1* , *position2* >

As an example,

```
box
arrow right from 1/3 of the way between last box.ne and last box.se
arrow right from 2/3 <last box.ne, last box.se>
```

produces



Naturally, the distance given by *fraction* can be greater than 1 or less than 0.

6. Variables and Expressions

It's generally a bad idea to write everything in absolute coordinates if you are likely to change things. PIC variables let you parameterize your picture:

```
a = 0.5;  b = 1

box wid a ht b
ellipse wid a/2 ht 1.5*b
move to Box1 - (a/2, b/2)
```

Expressions may use the standard operators +, -, *, /, and %, and parentheses for grouping.

Probably the most important variables are the predefined ones for controlling the default sizes of objects, listed in Section 3. These may be set at any time in any picture, and retain their values until reset.

You can use the height, width, radius, and x and y coordinates of any object or corner in an expression:

```
Box1.x              # the x coordinate of Box1
Box1.ne.y           # the y coordinate of the northeast corner of Box1
Box1.wid            # the width of Box1
Box1.ht             # and its height
2nd last circle.rad # the radius of the 2nd last circle
```

Any pair of expressions enclosed in parentheses defines a position; furthermore such positions can be added or subtracted to yield new positions:

(x, y)
$(x_1, y_1) + (x_2, y_2)$

If p_1 and p_2 are positions, then

(p_1, p_2)

refers to the point

$(p_1.x, p_2.y)$

7. More on Text

Normally, text is centered at the geometric center of the object it is associated with. The attribute *ljust* causes the left end to be at the specified point (which means that the text lies to the right of the specified place!), and *rjust* puts the right end at the place. *above* and *below* center the text one half line space in the given direction.

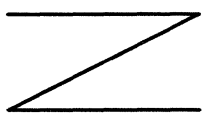
At the moment you can *not* compound text attributes: however natural it might seem, it is illegal to say "..." *above ljust*. This will be fixed eventually.

Text is most often an attribute of some other object, but you can also have self-standing text:

```
"this is some text" at 1,2 ljust
```

8. Lines and Splines

A "line" may actually be a path, that is, it may consist of connected segments like this:



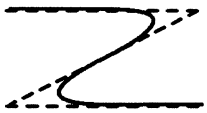
This line was produced by

```
line right 1i then down .5i left 1i then right 1i
```

A spline is a smooth curve guided by a set of straight lines just like the line above. It begins at the same place, ends at the same place, and in between is tangent to the mid-point of each guiding line. The syntax for a spline is identical to a (path) line except for using *spline* instead of *line*. Thus:

```
line dashed right 1i then down .5i left 1i then right 1i
spline from start of last line \
  right 1i then down .5i left 1i then right 1i
```

produces



(Long input lines can be split by ending each piece with a backslash.)

The elements of a path, whether for line or spline, are specified as a series of points, either in absolute terms or by *up*, *down*, etc. If necessary to disambiguate, the word *then* can be used to separate components, as in

```
spline right then up then left then up
```

which is not the same as

```
spline right up left up
```

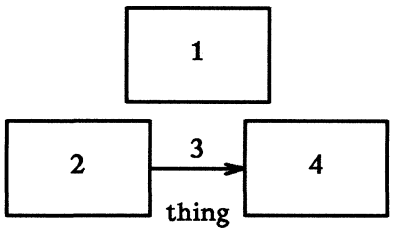
At the moment, arrowheads may only be put on the ends of a line or spline; splines may not be dotted or dashed.

9. Blocks

Any sequence of PIC statements may be enclosed in brackets *[...]* to form a block, which can then be treated as a single object, and manipulated rather like an ordinary box. For example, if we say

```
box "1"
[ box "2"; arrow "3" above; box "4" ] with .n at last box.s - (0,0.1)
"thing" at last [].s
```

we get



Notice that "last"-type constructs treat blocks as a unit and don't look inside for objects: *"last box.s"* refers to box 1, not box 2 or 4. You can use *last [ ]*, etc., just like *last box*.

Blocks have the same compass corners as boxes (determined by the bounding box). It is also possible to position a block by placing either an absolute coordinate (like *0,0*) or an internal label (like *A*) at some external point, as in

```
[ ...; A: ...; ... ] with .A at ...
```

Blocks join with other things like boxes do (i.e., at the center of the appropriate side).

It's not clear that this is the right thing to do, so it may change.

Names of variables and places within a block are local to that block, and thus do not affect variables and places of the same name outside. You can get at the internal place names with constructs like

```
last [].A
```

or

```
B.A
```

where *B* is a name attached to a block like so:

```
B : [ ... ;  A: ...;  ]
```

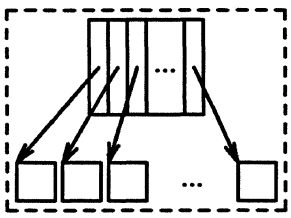
When combined with *define* statements (next section), blocks provide a reasonable simulation of a procedure mechanism.

Although blocks nest, it is currently possible to look only one level deep with constructs like *B.A*, although *A* may be further qualified (i.e., *B.A.sw* or *top of B.A* are legal).

The following example illustrates most of the points made above about how blocks work:

```
h = .5i
dh = .02i
dw = .1i
[
        Ptr: [
                boxht = h; boxwid = dw
                A: box
                B: box
                C: box
                box wid 2*boxwid "..."
                D: box
        ]
        Block: [
                boxht = 2*dw; boxwid = 2*dw
                movewid = 2*dh
                A: box; move
                B: box; move
                C: box; move
                box invis "..." wid 2*boxwid; move
                D: box
        ] with .t at Ptr.s - (0,h/2)
        arrow from Ptr.A to Block.A.nw
        arrow from Ptr.B to Block.B.nw
        arrow from Ptr.C to Block.C.nw
        arrow from Ptr.D to Block.D.nw
]
box dashed ht last [].ht+dw wid last [].wid+dw at last []
```

This produces



10. Macros

PIC provides a rudimentary macro facility, the simple form of which is identical to that in EQN:

```
define name X replacement text X
```

defines *name* to be the *replacement text*; *X* is any character that does not appear in the replacement. Any subsequent occurrence of *name* will be replaced by *replacement text*.

Macros with arguments are also available. The replacement text of a macro definition may contain occurrences of *$1* through *$9*; these will be replaced by the corresponding actual arguments when the macro is invoked. The invocation for a macro with arguments is

```
name(arg1, arg2, ...)
```

Non-existent arguments are replaced by null strings.

As an example, one might define a *square* by

```
define square X box ht $1 wid $1  $2 X
```

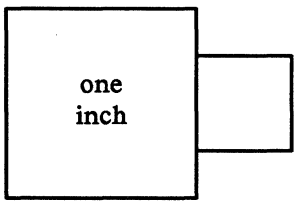
Then

```
square(1i, "one" "inch")
```

calls for a one inch square with the obvious label, and

```
square(0.5i)
```

calls for a square with no label:



Coordinates like *x,y* may be enclosed in parentheses, as in (*x,y*), so they can be included in a macro argument.

11. TROFF Interface

PIC is usually run as a TROFF preprocessor:

```
pic file | troff -ms
```

Run it before EQN and TBL if they are also present.

If the *.PS* line looks like

```
.PS <file
```

then the contents of *file* are inserted in place of the *.PS* line (whether or not the file contains *.PS* or *.PE*).

Other than this file inclusion facility, PIC copies the *.PS* and *.PE* lines from input to output intact, except that it adds two things right on the same line as the *.PS*:

```
.PS h w
```

h and *w* are the picture height and width in units. The *-ms* macro package has simple definitions for *.PS* and *.PE* that cause pictures to be centered and offset a bit from surrounding text.

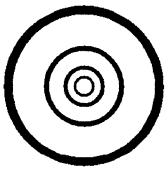
If "*.PF*" is used instead of *.PE*, the position after printing is restored to where it was

before the picture started, instead of being at the bottom. ("*F*" is for "flyback.")

Any input line that begins with a period is assumed to be a TROFF command that makes sense at that point; it is copied to the output at that point in the document. It is asking for trouble to add spaces or in any way fiddle with the line spacing here, but point size and font changes are generally harmless. So, for example,

```
.ps 24
circle radius .4i at 0,0
.ps 12
circle radius .2i at 0,0
.ps 8
circle radius .1i at 0,0
.ps 6
circle radius .05i at 0,0
.ps 10\" don't forget to restore size
```

gives



It is also safe to muck about with sizes and fonts and local motions within quoted strings ("...") in PIC, so long as whatever changes are made are unmade before exiting the string. For example, to print text in Old English in size 8, use

```
ellipse "\s8\f(OESmile!\fP\s0"
```

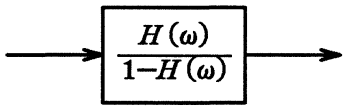
This produces

This is essentially the same rule as applies in EQN.

There is a subtle problem with complicated equations inside PIC pictures — they come out wrong if EQN has to leave extra vertical space for the equation. If your equation involves more than subscripts and superscripts, you must add to the beginning of each equation the extra information *space 0*:

```
arrow
box "$space 0 {H( omega )} over {1 - H( omega )}$"
arrow
```

This produces

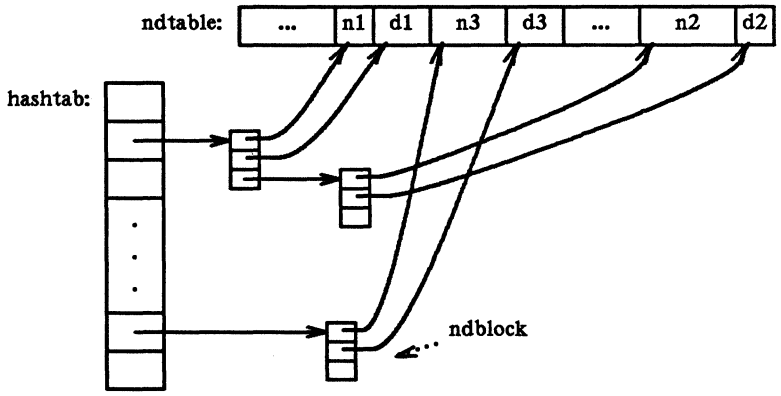


PIC normally generates commands for a new version of TROFF that has operators for drawing graphical objects like lines, circles, and so on. As distributed, PIC assumes that its output is going to the Mergenthaler Linotron 202 unless told otherwise with the *-T* option. At present, the other alternatives are *-Tcat* (the Graphic Systems CAT, which does slanted lines and curves badly) and *-Taps* (the Autologic APS-5). It is likely that the option will already have been set to the proper default for your system, unless you have a choice of

typesetters.

12. Some Examples

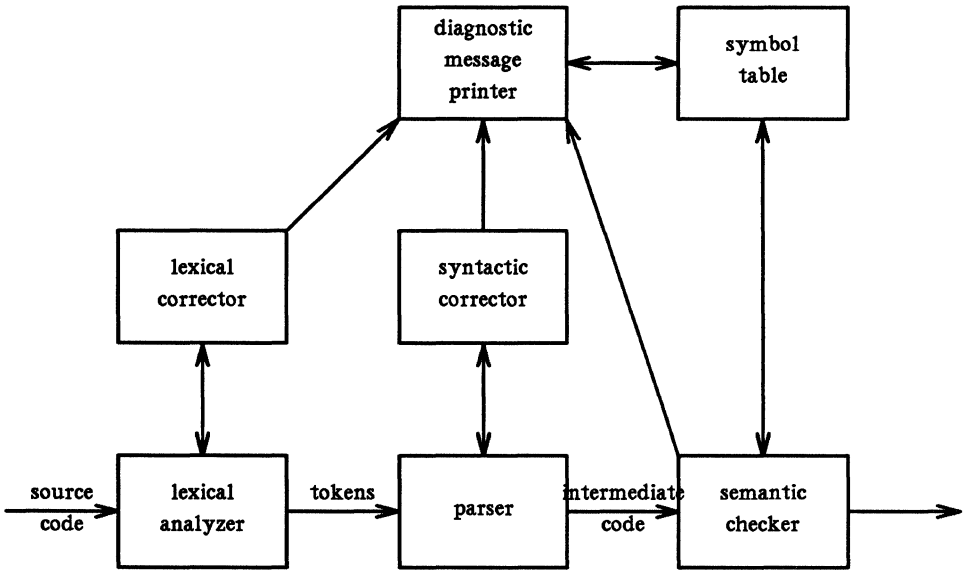
Herewith a handful of larger examples:



The input for the picture above was:

```
define ndblock X
        box wid boxwid/2 ht boxht/2
        down;  box same with .t at bottom of last box;   box same
X
boxht = .2i; boxwid = .3i; circlerad = .3i
down; box; box; box; box ht 3*boxht fi.ffi.ffi.fi
L: box; box; box invis wid 2*boxwid fihashtab:fiwith .e at 1st box .w
right
Start: box wid .5i with .sw at 1st box.ne + (.4i,.2i) fi...fi
N1: box wid .2i fin1fi;D1: box wid .3i fid1fi
N3: box wid .4i fin3fi;D3: box wid .3i fid3fi
box wid .4i fi...fi
N2: box wid .5i fin2fi;D2: box wid .2i fid2fi
arrow right from 2nd box
ndblock
spline -fi  right .2i from 3rd last box then to N1.sw + (0.05i,0)
spline -fi  right .3i from 2nd last box then to D1.sw + (0.05i,0)
arrow right from last box
ndblock
spline -fi  right .2i from 3rd last box to N2.sw-(0.05i,.2i) to N2.sw+(0.05i,0)
spline -fi  right .3i from 2nd last box to D2.sw-(0.05i,.2i) to D2.sw+(0.05i,0)
arrow right 2*linewid from L
ndblock
spline -fi  right .2i from 3rd last box to N3.sw + (0.05i,0)
spline -fi  right .3i from 2nd last box to D3.sw + (0.05i,0)
circle invis findblockfiat last box.e + (.7i,.2i)
arrow dotted from last circle to last box chop
box invis wid 2*boxwid findtable:fiwith .e at Start.w
```

This is the second example:



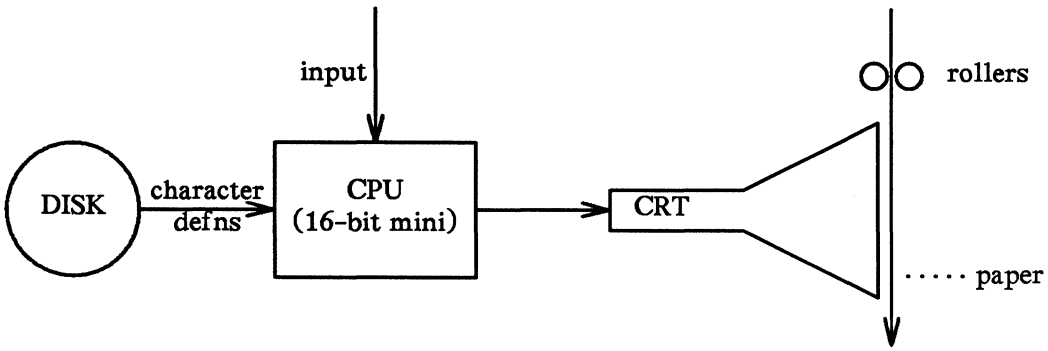
This is the input for the picture:

```
.PS 5
.ps 8
        arrow "source" "code"
LA:     box "lexical" "analyzer"
        arrow "tokens" above
P:      box "parser"
        arrow "intermediate" "code"
Sem:    box "semantic" "checker"
        arrow

        arrow <-> up from top of LA
LC:     box "lexical" "corrector"
        arrow <-> up from top of P
Syn:    box "syntactic" "corrector"
        arrow up
DMP:    box "diagnostic" "message" "printer"
        arrow <-> right  from right of DMP
ST:     box "symbol" "table"
        arrow from LC.ne to DMP.sw
        arrow from Sem.nw to DMP.se
        arrow <-> from Sem.top to ST.bot
.PE
```

There are eighteen objects (boxes and arrows) in the picture, and one line of PIC input for each; this seems like an acceptable level of verbosity.

The next example is the following:



Basic Digital Typesetter

This is the input for example 3:

```
.KS
.PS 5i
circle "DISK"
arrow "character" "defns"
box "CPU" "(16-bit mini)"
{ arrow <- from top of last box up "input " rjust }
arrow
CRT: "  CRT" ljust
line from CRT - 0,0.075 up 0.15 \
then right 0.5 \
then right 0.5 up 0.25 \
then down 0.5+0.15 \
then left 0.5 up 0.25 \
then left 0.5

Paper: CRT + 1.0+0.05,0
arrow from Paper + 0,0.75 to Paper - 0,0.5
{ move to start of last arrow down 0.25
  { move left 0.015; circle rad 0.05 }
  { move right 0.015; circle rad 0.05; "  rollers" ljust }
}
"  paper" ljust at end of last arrow right 0.25 up 0.25
line left 0.2 dotted
.PE
.ce
Basic Digital Typesetter
.sp
.KE
```

13. Final Observations

PIC is not a sophisticated tool. The fundamental approach — Cartesian coordinates and real measurements — is not the easiest thing in the world to work with, although it does have the merit of being in some sense sufficient. Much of the syntactic sugar (or corn syrup) — corners, joining things implicitly, etc. — is aimed at making positioning and sizing automatic, or at least relative to previous things, rather than explicit.

Nonetheless, PIC does seem to offer some positive values. Most notably, it is integrated with the rest of the standard Unix document preparation software. In particular, it positions text correctly in relation to graphical objects; this is not true of any of the interactive graphical editors that I am aware of. It can even deal with equations in a

natural manner, modulo the *space 0* nonsense alluded to above.

A standard question is, "Wouldn't it be better if it were interactive?" The answer seems to be both yes and no. If one has a decent input device (which I do not), interaction is certainly better for sketching out a figure. But if one has only standard terminals (at home, for instance), then a linear representation of a figure is better. Furthermore, it is possible to generate PIC input from a program: I have used AWK[9] to extract numbers from a report and generate the PIC commands to make histograms. This is hard to imagine with most of the interactive systems I know of.

In any case, the issue is far from settled; comments and suggestions are welcome.

Acknowledgements

I am indebted to Chris Van Wyk for ideas from several versions of IDEAL. He and Doug McIlroy have also contributed algorithms for line and circle drawing, and made useful suggestions on the design of PIC. Theo Pavlidis contributed the basic spline algorithm. Charles Wetherell pointed out reference [2] to me, and made several valuable criticisms on an early draft of the language and manual. The exposition in this version has been greatly improved by suggestions from Jim Blinn. I am grateful to my early users — Brenda Baker, Dottie Luciani, and Paul Tukey — for their suggestions and cheerful use of an often shaky and clumsy system.

References

1. J. F. Ossanna, "NROFF/TROFF User's Manual," *UNIX Programmer's Manual* **2**, Section 22 (January 1979).
2. Brian W. Kernighan and Lorinda L. Cherry, "A System for Typesetting Mathematics," *Communications of the ACM* **18**(3), pp. 151-157 (1975).
3. M. E. Lesk, "Tbl — A Program to Format Tables," *UNIX Programmer's Manual* **2**, Section 10 (January 1979).
4. M. E. Lesk, "Some Applications of Inverted Indexes on the UNIX System," *UNIX Programmer's Manual* **2**, Section 11 (January 1979).
5. Christopher J. Van Wyk and C. J. Van Wyk, "A Graphics Typesetting Language," *SIGPLAN Symposium on Text Manipulation*, Portland, Oregon (June, 1981).
6. John C. Beatty, "PICTURE — A picture-drawing language for the Trix/Red Report Editor," Lawrence Livermore Laboratory Report UCID-30156 (April 1977).
7. Anon., "V — A viewgraph generating language," Bell Laboratories internal memorandum (May 1979).
8. B. W. Kernighan, "PIC — A Language for Typesetting Graphics," *Software Practice & Experience* **12**(1), pp. 1-21 (January, 1982).
9. A. V. Aho, P. J. Weinberger, and B. W. Kernighan, "AWK - A Pattern Scanning and Processing Language," *Software Practice and Experience* **9**, pp. 267-280 (April 1979).

Appendix A: PIC Reference Manual

Pictures

The top-level object in PIC is the "picture":

picture:
.PS *optional-width*
element-list
.PE

If *optional-width* is present, the picture is made that many inches wide, regardless of any dimensions used internally. The height is scaled in the same proportion.

If instead the line is

.PS <f

the file *f* is inserted in place of the *.PS* line.

If *.PF* is used instead of *.PE*, the position after printing is restored to what it was upon entry.

Elements

An *element-list* is a list of elements (what else?); the elements are

element:
primitive attribute-list
placename : *element*
placename : *position*
variable = *expression*
direction
troff-command
{ *element-list* }
[*element-list*]

Elements in a list must be separated by newlines or semicolons; a long element may be continued by ending the line with a backslash. Comments are introduced by a # and terminated by a newline.

Variable names begin with a lower case letter; place names begin with upper case. Place and variable names retain their values from one picture to the next.

The current position and direction of motion are saved upon entry to a {...} block and restored upon exit.

Elements within a block enclosed in *[...]* are treated as a unit; the dimensions are determined by the extreme points of the contained objects. Names, variables, and direction of motion within a block are local to that block.

troff-command is any line that begins with a period. Such lines are assumed to make sense in the context where they appear; accordingly, if it doesn't work, don't call.

Primitives

The primitive objects are

primitive:
box
circle
ellipse
arc
line
arrow
move
spline
"any text at all"

arrow is a synonym for *line* ->.

Attributes

An *attribute-list* is a sequence of zero or more attributes; each attribute consists of a keyword, perhaps followed by a value. In the following, *e* is an expression and *opt-e* an optional expression.

attribute:

h(eigh)t *e*	wid(th) *e*
rad(ius) *e*	diam(eter) *e*
up *opt-e*	down *opt-e*
right *opt-e*	left *opt-e*
from *position*	to *position*
at *position*	with *corner*
by *e*, *e*	then
dotted *opt-e*	dashed *opt-e*
chop *opt-e*	-> <- <->
same	invis
text-list	

Missing attributes and values are filled in from defaults. Not all attributes make sense for all primitives; irrelevant ones are silently ignored. These are the currently meaningful attributes:

box:
height, width, at, dotted, dashed, invis, same, *text*
circle and ellipse:
radius, diameter, height, width, at, invis, same, *text*
arc:
up, down, left, right, height, width, from, to, at, radius,
invis, same, cw, <-, ->, <->, *text*
line, arrow
up, down, left, right, height, width, from, to, by, then,
dotted, dashed, invis, same, <-, ->, <->, *text*
spline:
up, down, left, right, height, width, from, to, by, then,
invis, <-, ->, <->, *text*
move:
up, down, left, right, to, by, same, *text*
"text...":
at, *text*

The attribute *at* implies placing the geometrical center at the specified place. For lines, splines and arcs, *height* and *width* refer to arrowhead size.

Text

Text is normally an attribute of some primitive; by default it is placed at the geometrical center of the object. Stand-alone text is also permitted. A *text-list* is a list of text items; a text item is a quoted string optionally followed by a positioning request:

text-item:
"..."
"..." center
"..." ljust
"..." rjust
"..." above
"..." below

If there are multiple text items for some primitive, they are centered vertically except as qualified. Positioning requests apply to each item independently.

Text items can contain TROFF commands for size and font changes, local motions, etc., but make sure that these are balanced so that the entering state is restored before exiting.

Positions and places

A position is ultimately an *x,y* coordinate pair, but it may be specified in other ways.

position:
e, e
place ± *e, e*
(*position, position*)
e [of the way] between *position* and *position*
e < *position , position* >

The pair *e, e* may be enclosed in parentheses.

place:
placename *optional-corner*
corner placename
Here
corner of *nth primitive*
nth primitive optional-corner

A *corner* is one of the eight compass points or the center or the start or end of a primitive. (Not text!)

corner:
.n .e .w .s .ne .se .nw .sw
.t .b .r .l
.c .start .end

Each object in a picture has an ordinal number; *nth* refers to this.

nth:
*n*th
*n*th last

Since barbarisms like *1th* are barbaric, synonyms like *1st* and *3st* are accepted as well.

Variables

The built-in variables and their default values are:

boxwid 0.75i	boxht 0.5i
circlerad 0.25i	
ellipsewid 0.75i	ellipseht 0.5i
arcrad 0.25i	
linewid 0.5i	lineht 0.5i
movewid 0.5i	movewid 0.5i
arrowht 0.1i	arrowwid 0.05i
dashwid 0.1i	
scale 1	

These may be changed at any time, and the new values remain in force until changed again. Dimensions are divided by *scale* during output.

Expressions

Expressions in PIC are evaluated in floating point. All numbers representing dimensions are taken to be in inches.

expression:
- *e* + *e*
- *e* - *e*
- *e* * *e*
- *e* / *e*
- *e* % *e* (modulus)
- - *e*
- (*e*)
- variable
- number
- *place* .x
- *place* .y
- *place* .ht
- *place* .wid
- *place* .rad

Definitions

The *define* statement is not part of the grammar.

define:
- define *name* X *replacement text* X

Occurrences of *$1* through *$9* in the replacement text will be replaced by the corresponding arguments if *name* is invoked as

name(*arg1*, *arg2*, ...)

Non-existent arguments are replaced by null strings. *Replacement text* may contain newlines.