Additional Material

This section contains additional supplementary material, of use to users at Stanford, that is not part of the Unix manual supplied by Berkeley.

Changes from 4.1BSD to 4.2BSD Vax Unix at Stanford University

14 September 1984

There have been many changes made to Unix for the version known as "4.2BSD Unix". In addition, some of the modifications made at Stanford to the previous version of Unix (4.1BSD) have themselves been changed. This document, together with the attached document "Bug Fixes and Changes in 4.2BSD", should answer most of your questions.

We do not list all the changes, merely the ones which might confuse the average user. Please assume that other changes you encounter are not bugs, at least until you read the manual page.

Programs whose names or behaviours have changed

stty	Formerly,the *alterase* option took no arguments; now it does. Please check your .login files for this; for example, you probably will want a line in your .login file like: `stty new crt alterase ↑h`
telnet	Now called **puptelnet**.
iptelnet	Now called **telnet**, and has a slightly different (some might say worse) command interface.
ftp	Now called **pupftp**.
ipftp	Now called **ftp**, and has a different command interface.
talk	A completely different program, much fancier.
echosend	Now called **pupecho**.
backup	Works better, can handle more than one file at a time.

Note: If you use a Sun workstation as a telnet terminal, this is no longer terminal type "sun" (which is now reserved for Sun Microsystems software.) The proper incantation (for csh users) is

```
    % setenv TERM vgts
```

The "vgts" terminal type comes in a variety of optional sizes, notably "vgts52" for large windows.

Programs that no longer exist

nwrite	Use **talk**, **write**, or **wall**
spice	Some of the source files have been mangled.
cifplot	Not yet converted to 4.2BSD.
twunix	Source files were lost years ago.
pgrind	Subsumed by changes to **vgrind** (not tested)
nfmt	Not converted to 4.2BSD.

Note that the "CMU IPC facility" which had been part of the kernel is no longer available.

New programs

gripe	Mails a bug report to the support staff.
cnest	Checks C source code to find improperly nested comments.
ingroup	List members of a protection group.
linelen	Shows how long lines in a text file are.
symlchk	Checks for incorrect symbolic links.
buildmake, makedep	A pair of programs useful for maintaining makefiles.
cparen	Helps verify parenthesization of C expressions.
dtree	Prints fancy picture of directory structures.
btroff	Crude **troff** output to the Boise printer (doesn't really work)
rtar	Allows **tar** to be used with a tape drive on a different machine.
shar	Takes a group of text files and produces a shell script that, when run, recreates them. Useful for mailing a set of files to someone.
rdist	Complicated program used to distribute files automatically to other machines.

Ex Quick Reference

Entering/leaving ex

% ex *name*	edit *name*, start at end
% ex +*n name*	... at line *n*
% ex −t *tag*	start at *tag*
% ex −r	list saved files
% ex −r *name*	recover file *name*
% ex *name* ...	edit first; rest via :n
% ex −R *name*	read only mode
: x	exit, saving changes
: q!	exit, discarding changes

Ex states

Command	Normal and initial state. Input prompted for by :. Your kill character cancels partial command.
Insert	Entered by **a i** and **c**. Arbitrary text then terminates with line having only **.** character on it or abnormally with interrupt.
Open/visual	Entered by **open** or **vi**, terminates with Q or ^\\.

Ex commands

abbrev	ab	next	n	unabbrev	una
append	a	number	nu	undo	u
args	ar	open	o	unmap	unm
change	c	preserve	pre	version	ve
copy	co	print	p	visual	vi
delete	d	put	pu	write	w
edit	e	quit	q	xit	x
file	f	read	re	yank	ya
global	g	recover	rec	*window*	z
insert	i	rewind	rew	*escape*	!
join	j	set	se	*lshift*	<
list	l	shell	sh	*print next*	CR
map		source	so	*resubst*	&
mark	ma	stop	st	*rshift*	>
move	m	substitute	s	*scroll*	^D

Ex command addresses

n	line *n*	/*pat*	next with *pat*
.	current	?*pat*	previous with *pat*
$	last	*x*-*n*	*n* before *x*
+	next	*x*,*y*	*x* through *y*
−	previous	′*x*	marked with *x*
+*n*	*n* forward	″	previous context
[illegible]	[illegible]		

Specifying terminal type

% setenv TERM *type*	*csh* and all version 6
$ TERM=*type*; export TERM	*sh* in Version 7
See also *tset*(1)	

Some terminal types

2621	43	adm31	dw1	h19
2645	733	adm3a	dw2	i100
300s	745	c100	gt40	mime
33	act4	dm1520	gt42	owl
37	act5	dm2500	h1500	t1061
4014	adm3	dm3025	h1510	vt52

Initializing options

EXINIT	place **set**'s here in environment var.
set *x*	enable option
set no*x*	disable option
set *x*=*val*	give value *val*
set	show changed options
set *x*?	show value of option *x*

Useful options

autoindent	ai	supply indent
autowrite	aw	write before changing files
ignorecase	ic	in scanning
lisp		( ) { } are s-exp's
list		print ^I for tab, $ at end
magic		. [* special in patterns
number	nu	number lines
paragraphs	para	macro names which start ...
redraw		simulate smart terminal
scroll		command mode lines
sections	sect	macro names ...
shiftwidth	sw	for < >, and input ^D
showmatch	sm	to) and } as typed
slowopen	slow	choke updates during insert
window		visual mode lines
wrapscan	ws	around end of buffer?
wrapmargin	wm	automatic line splitting

Scanning pattern formation

↑	beginning of line
$	end of line
.	any character
`\<`	beginning of word
`\>`	end of word
[*str*]	any char in *str*
[↑*str*]	... not in *str*

Vi Quick Reference

Entering/leaving vi

% vi *name*	edit *name* at top
% vi +*n name*	... at line *n*
% vi + *name*	... at end
% vi −r	list saved files
% vi −r *name*	recover file *name*
% vi *name* ...	edit first; rest via :n
% vi −t *tag*	start at *tag*
% vi +/*pat name*	search for *pat*
% view *name*	read only mode
ZZ	exit from vi, saving changes
^Z	stop vi for later resumption

The display

Last line	Error messages, echoing input to : / ? and !, feedback about i/o and large changes.
@ lines	On screen only, not in file.
~ lines	Lines past end of file.
^*x*	Control characters, ^? is delete.
tabs	Expand to spaces, cursor at last.

Vi states

Command	Normal and initial state. Others return here. ESC (escape) cancels partial command.
Insert	Entered by **a i A I o O c C s S R**. Arbitrary text then terminates with ESC character, or abnormally with interrupt.
Last line	Reading input for : / ? or !; terminate with ESC or CR to execute, interrupt to cancel.

Counts before vi commands

line/column number	z G |
scroll amount	^D ^U
replicate insert	a i A I
repeat effect	most rest

Simple commands

dw	delete a word
de	... leaving punctuation
dd	delete a line
3dd	... 3 lines
i*text*ESC	insert text *abc*
[illegible]	[illegible]

ESC	end insert or incomplete cmd
^?	(delete or rubout) interrupts
^L	reprint screen if ^? scrambles it

File manipulation

:w	write back changes
:wq	write and quit
:q	quit
:q!	quit, discard changes
:e *name*	edit file *name*
:e!	reedit, discard changes
:e + *name*	edit, starting at end
:e +*n*	edit starting at line *n*
:e #	edit alternate file
^↑	synonym for :e #
:w *name*	write file *name*
:w! *name*	overwrite file *name*
:sh	run shell, then return
:!*cmd*	run *cmd*, then return
:n	edit next file in arglist
:n *args*	specify new arglist
:f	show current file and line
^G	synonym for :f
:ta *tag*	to tag file entry *tag*
^]	:ta, following word is *tag*

Positioning within file

^F	forward screenfull
^B	backward screenfull
^D	scroll down half screen
^U	scroll up half screen
G	goto line (end default)
/*pat*	next line matching *pat*
?*pat*	prev line matching *pat*
n	repeat last / or ?
N	reverse last / or ?
/*pat*/+*n*	n'th line after *pat*
?*pat*?−*n*	n'th line before *pat*
]]	next section/function
[[	previous section/function
%	find matching ( ) { or }

Adjusting the screen

^L	clear and redraw
^R	retype, eliminate @ lines
zCR	redraw, current at window top
z−	... at bottom
z.	... at center
/*pat*/z−	*pat* line at bottom
z*n*.	use *n* line window
^E	scroll window down 1 line
^Y	scroll window up 1 line

''	... at first non-white in line
m*x*	mark position with letter *x*
`*x*	to mark *x*
'*x*	... at first non-white in line

Line positioning

H	home window line
L	last window line
M	middle window line
+	next line, at first non-white
−	previous line, at first non-white
CR	return, same as +
↓ or j	next line, same column
↑ or k	previous line, same column

Character positioning

↑	first non white
0	beginning of line
$	end of line
h or →	forward
l or ←	backwards
^H	same as ←
space	same as →
f*x*	find *x* forward
F*x*	f backward
t*x*	upto *x* forward
T*x*	back upto *x*
;	repeat last f F t or T
,	inverse of ;
|	to specified column
%	find matching ({) or }

Words, sentences, paragraphs

w	word forward
b	back word
e	end of word
)	to next sentence
}	to next paragraph
(	back sentence
{	back paragraph
W	blank delimited word
B	back W
E	to end of W

Commands for LISP

)	Forward s-expression
}	... but don't stop at atoms
(	Back s-expression
{	... but don't stop at atoms

^W	erases last word
erase	your erase, same as ^H
kill	your kill, erase input this line
\	escapes ^H, your erase and kill
ESC	ends insertion, back to command
^?	interrupt, terminates insert
^D	backtab over *autoindent*
↑^D	kill *autoindent*, save for next
0^D	... but at margin next also
^V	quote non-printing character

Insert and replace

a	append after cursor
i	insert before
A	append at end of line
I	insert before first non-blank
o	open line below
O	open above
r*x*	replace single char with *x*
R	replace characters

Operators (double to affect lines)

d	delete
c	change
<	left shift
>	right shift
!	filter through command
=	indent for LISP
y	yank lines to buffer

Miscellaneous operations

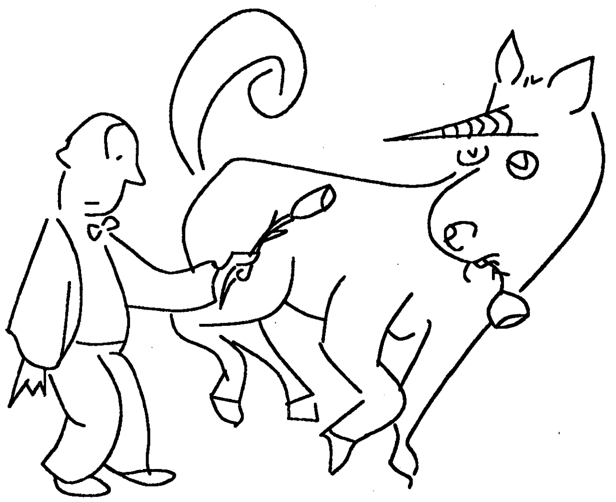
C	change rest of line
D	delete rest of line
s	substitute chars
S	substitute lines
J	join lines
x	delete characters
X	... before cursor
Y	yank lines

Yank and put

p	put back lines
P	put before
"*x*p	put from buffer *x*
"*x*y	yank to buffer *x*
"*x*d	delete into buffer *x*

Undo, redo, retrieve

u	undo last change
U	restore current line
.	repeat last change
"*d*p	retrieve *d*'th last delete



Unix Emacs

James Gosling @ CMU
May, 1982

1. Introduction

"What is EMACS? It is a tree falling in the forest with no one to hear it. It is a beautiful flower that smells awful."

This manual attempts to describe the Unix implementation of EMACS, an extensible display editor. It is an *editor* in that it is primarily used for typing in and modifying documents, programs, or anything else that is represented as text. It uses a *display* to interact with the user, always keeping an accurate representation of what is happening visible on the screen that changes in step with the changes made to the document. The feature that distinguishes EMACS from most other editors is its *extensibility*, that is, a user of EMACS can dynamically change EMACS to suit his own tastes and needs.

Calling this editor EMACS is rather presumptuous and even dangerous. There are two major editors called EMACS. The first was written at MIT for their ITS systems as an extension to TECO. This editor is the spiritual father of all the EMACS-like editors; it's principal author was Richard Stallman. The other was also written at MIT, but it was written in MacLisp for Multics by Bernie Greenberg. This editor picks up where ITS EMACS leaves off in terms of its extension facilities. Unix EMACS was called EMACS in the hope that the cries of outrage would be enough to goad the author and others to bring it up to the standards of what has come before.

This manual is organized in a rather haphazard manner. The first several sections were written hastily in an attempt to provide a general introduction to the commands in EMACS and to try to show the method in the madness that is the EMACS command structure. Section 21 (page 39) contains a complete but concise description of all the commands and is in alphabetical order based on the name of the command. Preceding sections generally do not give a complete description of each command, rather they give either the name of the command or the key to which the command is conventionally bound. Section 22 (page 81) lists for each key the command to which it is conventionally bound. The options which may be set with the *set* command are described in section 22, (page 76).

2. The Screen

EMACS divides a screen into several areas called *windows*, at the bottom of the screen there is a one line area that is used for messages and questions from EMACS. Most people will only be using one window, at least until they become more familiar with EMACS. A window is displayed as a set of lines, at the bottom of each window is its *mode line* (For more information on mode lines see section 17, page 19). The lines above the mode line contain an image of the text you are editing in the region around *dot* (or *point*). Dot is the reference around which editing takes place. Dot is a pointer which points at a position *between* two characters. On the screen, the cursor will be positioned on the character that immediatly follows dot. When characters are inserted, they are inserted at the position where dot points; commands exist that delete characters both to the left and to the right of dot. The text on the screen always reflects they way that the text looks *now*.

3. Input Conventions

Throughout this manual, characters which are used as commands are printed in bold face: **X**. They will sometimes have a *control* prefix which is printed as an uparrow character: **↑X** is control-X and is typed by holding down the control (often labeled *ctrl* on the top of the key) and simultaneously striking X. Some will have an *escape* (sometimes called *meta*) prefix which is usually printed thus: **ESC-X** and typed by striking the escape key (often labeled *esc*) then X. And some will have a **↑X** prefix which is printed **↑XX** which is typed by holding down the control key, striking X, releasing the control key then striking X again.

For example, **ESC-↑J** is typed by striking ESC then holding down the control key and striking J.

4. Invoking EMACS

EMACS is invoked as a Unix command by typing

```
    emacs files
```

to the Shell (the Unix command interpreter). EMACS will start up, editing the named files. You will probably only want to name one file. If you don't specify any names, EMACS will use the same names that it was given the last time that it was invoked. Gory details on the invocation of EMACS can be found in section 13.4, page 14.

5. Basic Commands

Normally each character you type is interpreted individually by EMACS as a command. The instant you type a character the command it represents is performed immediatly.

All of the normal printing characters when struck just insert themselves into the buffer at dot.

To move dot there are several simple commands. **↑F** moves dot forward one character, **↑B** moves it backward one character. **↑N** moves dot to the same column on the next line, **↑P** moves it to the same column on the previous line.

String searches may be used to move dot by using the **↑S** command to search in the forward direction and **↑R** to search in the reverse direction.

Deletions may be performed using **↑H** (*backspace*) to delete the character to the left of dot and **↑D** to delete the character to the right of dot.

The following table summarizes all of the motion and deletion commands.

	Direction			
	Move		Delete	
Units of Motion	Left	Right	Left	Right
Characters	**↑B**	**↑F**	**↑H**	**↑D**
Words	**ESC-B**	**ESC-F**	**ESC-H**	**ESC-D**
Intra line	**↑A**	**↑E**		**↑K**
Inter line	**↑P**	**↑N**		

6. Unbound Commands

Even though the number of characters available to use for EMACS commands is large, there are still more commands than characters. You probably wouldn't want to bind them all to keys even if you could. Each command has a long name and by that long name may be bound to a key. For example, **↑F** is normally bound to the command named *forward-character* which moves dot forward one character.

There are many commands that are not normally bound to keys. These must be executed with the **ESC-X** command or by binding them to a key (via the bind-to-key command). Heaven help the twit who rebinds **ESC-X.**

The ESC-X command will print ": " on the last line of the display and expect you to type in the name of a command. Space and ESC characters may be struck to invoke Tenex style command completion (ie. you type in the first part of the command, hit the space bar, and EMACS will fill in the rest for you -- it will complain if it can't figure out what you're trying to say). If the command requires arguments, they will also be prompted for on the bottom line.

7. Getting Help

EMACS has many commands that let you ask EMACS for help about how to use EMACS. The simplest one is ESC-? (apropos) which asks you for a keyword and then displays a list of those commands whose full name contains the keyword as a substring. For example, to find out which commands are available for dealing with windows, type ESC-?, EMACS will ask "Keyword:" and you reply "window". A list like the following appears:

```
beginning-of-window      ESC-,
delete-other-windows     ↑X1
delete-window            ↑XD
end-of-window            ESC-.
enlarge-window           ↑XZ
line-to-top-of-window    ESC-!
next-window              ↑XN
page-next-window         ESC-↑V
previous-window          ↑XP
shrink-window            ↑X↑Z
split-current-window     ↑X2
```

To get detailed information about some command, the *describe-command* command can be used. It asks for the name of a command, then displays the long documentation for it from the manual. For example, if you wanted more information about the *shrink-window* command, just type "ESC-Xdescribe-command shrink-window" and EMACS will reply:

```
shrink-window                              ↑X↑Z
        Makes the current window one line shorter, and the window below
       (or the one above if there is no window below) one line taller.
       Can't be used if there is only one window on the screen.
```

If you want to find out what command is bound to a particular key, *describe-key* will do it for you. *Describe-bindings* can be used to make a "wall chart" description of the key bindings in the currently running EMACS, taking into account all of the bindings you have made.

8. Buffers and Windows

There are two fundamental objects in EMACS, *buffers* and *windows*. A buffer is a chunk of text that can be edited, it is often the body of a file. A window is a region on the screen through which a buffer may be viewed. A window looks at one buffer, but a buffer may be on view in several windows. It is often handy to have two windows looking at the same buffer so that you can be looking at two separate parts of the same file, for example, a set of declarations and a piece of code that uses those declarations. Similarly, it is often handy to have two different buffers on view in two windows.

The commands which deal with windows and buffers are: beginning-of-window (ESC-,), delete-other-windows (↑X1), delete-region-to-buffer (ESC-↑W), delete-window (↑XD), end-of-window (ESC-.), enlarge-window (↑XZ), line-to-top-of-window (ESC-!), list-buffers (↑X↑B), next-window (↑XN), page-next-window (ESC-↑V), previous-window (↑XP), shrink-window (↑X↑Z), split-current-window (↑X2), switch-to-buffer (↑XB), use-old-buffer (↑X↑O) and yank-buffer (ESC-↑Y). See the command description section for more details on each of these.

9. Terminal types

Grim reality being what it is, EMACS has to deal with a wide assortment of displays from many manufacturers. Each manufacturer has their own perverted idea of how programs should communicate with the display, so it is important for EMACS to correctly be told what type of terminal is being used. Under Unix, this is done by setting the environment variable 'TERM'. Normally, the operating system should set this to correspond to the type of terminal that you are using and you won't have to concern yourself with it. However, problems may arise and there are a few things that you should know.

'TERM' is a string variable whose value is the name of the type of terminal that you are using. If you are using the standard Unix shell then it should be set using the commands:

```
    TERM=...
    export TERM
```

If you're using the C shell (csh) then it should be set using the command:

```
    setenv TERM ...
```

where '...' is the appropriate terminal type. Consult your system administrator for a current list of valid terminal types. A good place to look is the file "/etc/termcap", it contains a list of all the terminals supported by EMACS. A few of the more common values are:

concept-lnz	For Concepts with the special firmware for EMACS.
concept	Concept 100, 104 and 108's from HDS.
h19	For Heathkit or Zenith model 19 terminals.
vt100	For VT100's from DEC, or any of the thousands of look-alikes.
aaa	For the Ann Arbor Ambassador.

10. Compiling programs

One of the most powerful features of Unix EMACS is the facility provided for compiling programs and coping with error messages from the compilers. It essential that you understand the standard Unix program *make* (even if you don't use EMACS). This program takes a database (a *makefile*) that describes the relationships among files and how to regenerate (recompile) them. If you have a program that is made up of many little pieces that have to be individually compiled and carefully crafted together into a single executable file, *make* can make your life orders of magnitude easier; it will automatically recompile only those pieces that need to be recompiled and put them together. EMACS has a set of commands that gracefully interact with this facility.

The ↑X↑E (*execute*) command writes all modified buffers and executes the *make* program. The output of *make* will be placed into a buffer called *Error log* which will be visible in some window on the screen. As soon as *make* has finished EMACS parses all of its output to find all the error messages and figure out the files and lines referred to. All of this information is squirreled away for later use by the ↑X↑N command.

The ↑X↑N (*next*) command takes the next error message from the set prepared by ↑X↑E and does three things with it:

- Makes the message itself visible at the top of a window. The buffer will be named *Error log.*
- Does a *visit* (see the ↑X↑V command) on the file in which the error occurred.
- Sets dot to the beginning of the line where the compiler saw the error. This setting of dot takes into account changes to the file that may have been made since the compilation was attempted. EMACS perfectly compensates for any changes that may have been made and always positions the text on the correct line (well, correct as far as the compiler was concerned; the compiler itself may have been a trifle confused about where the error occurred)

If you've seen all the error messages ↑X↑N will say so and do nothing else.

So, the general scenario for dealing with programs is:

- Build a *make* database to describe how your program is to be compiled.
- Compile your program from within EMACS by typing ↑X↑E.
- If there were errors, step through them by typing ↑X↑N, correcting the error, and typing ↑X↑N to get the next.
- When you run out of error messages, type ↑X↑E to try the compilation again.
- When you finally manage to get your beast to compile without any errors, type ↑C to say goodbye to EMACS.
- You'll probably want to use *sdb*, the symbolic debugger, to debug your program.

11. Dealing with collections of files

The ↑X↑E command doesn't always execute the *make* program: if it is given a non-zero argument it will prompt for a Unix command line to be executed in place of *make*. All of the other parts of ↑X↑E are unchanged, namely it still writes all modified buffers before executing the command and parses the output of the command execution for line numbers and file names.

This can be used in some very powerful ways. For example, consider the *grep* program. Typing "↑U↑X↑E`grep -n MyProc *.c`ESC" will scan all C programs in the current directory and look for all occurrences of the string "MyProc". After *grep* has finished you can use EMACS (via the ↑X↑N command) to examine and possibly change every instance of the string from a whole collection of files. This makes the task of changing all calls to a particular procedure much easier. **Note:** this only works with the version of *grep* in /usr/jag/bin which has been modified to print line numbers in a format that EMACS can understand.

There are many more uses. The *lint* program, for example. Scribe users might find "`cat MyReport.otl`" to be useful.

A file name/line number pair is just a string embedded someplace in the text of the error log that has the form "FileName`, line` LineNumber". The FileName may or may not be surrounded by quotes ("). The critical component is the string "`, line `" that comes between the file name and the line number. Roll your own file scanning programs, it can make your life much easier.

12. Abbrev mode

Abbrev mode allows the user to type abbreviations into a document and have EMACS automatically expand them. If you have an abbrev called "rhp" that has been defined to expand to the string "rhinocerous party" and have turned on abbrev mode then typing the first non-alphanumeric character after having typed "rhp" causes the string "rhp" to be replaced by "rhinocerous party". The capitalization of the typed in abbreviation controls the capitalization of the expansion: "Rhp" would expand as "Rhinocerous party" and "RHP" would expand as "Rhinocerous Party".

Abbreviations are defined in *abbrev tables*. There is a global abbrev table which is used regardless of which buffer you are in, and a local abbrev table which is selected on a buffer by buffer basis, generally depending on the major mode of the buffer.

Define-global-abbrev takes two arguments: the name of an abbreviation and the phrase that it is to expand to. The abbreviation will be defined in the global abbrev table. Define-local-abbrev is like define-global-abbrev except that it defines the abbreviation in the current local abbrev table.

The use-abbrev-table command is used to select (by name) which abbrev table is to be used locally in this buffer. The same abbrev table may be used in several buffers. The mode packages (like electric-c and text) all set up abbrev tables whose name matches the name of the mode.

The switch *abbrev-mode* must be turned on before EMACS will attempt to expand abbreviations. When abbrev-mode is turned on, the string "abbrev" appears in the mode section of the mode line for the buffer. Use-abbrev-table automatically turns on abbrev-mode if either the global or new local abbrev tables are non-empty.

All abbreviations currently defined can be written out to a file using the write-abbrev-file command. Such a file can be edited (if you wish) and later read back in to define the same abbreviations again. Read-abbrev-file reads in such a file and screams if it cannot be found, quietly-read-abbrev-file doesn't complain (it is primarily for use in startups so that you can load a current-directory dependant abbrev file without worrying about the case where the file doesn't exist).

People writing MLisp programs can have procedures invoked when an abbrev is triggered. Use the commands *define-hooked-global-abbrev* and *define-hooked-local-abbrev* to do this. These behave exactly as the unhooked versions do except that they also associate a named command with the abbrev. When the abbrev triggers, rather than replacing the abbreviation with the expansion phrase the hook procedure is invoked. The character that triged the abbrev will not have been inserted, but will be inserted immediatly after the hook procedure returns [unless the procedure returns 0]. The abbreviation will be the word immediatly to the left of dot, and the function *abbrev-expansion* returns the phrase that the abbrev would have expanded to.

13. Extensibility

Unix EMACS has two extension features: macros and a built in Lisp system. Macros are used when you have something quick and simple to do, Lisp is used when you want to build something fairly complicated like a new language dependant mode.

13.1. Macros

A *macro* is just a piece of text that EMACS remembers in a special way. When a macro is *executed* the characters that make up the macro are treated as though they had been typed at the keyboard. If you have some common sequence of keystrokes you can define a macro that contains them and instead of retyping them just call the macro. There are two ways of defining macros:

The easiest is called a *keyboard* macro. A keyboard macro is defined by typing the start-remembering command (↑X() then typing the commands which you want to have saved (which will be executed as you type them so that you can make sure that they are right) then typing the stop-remembering command (↑X)). To execute the keyboard macro just type the execute-keyboard-macro command (↑Xe). You can only have one keyboard macro at a time. If you define a new keyboard macro the old keyboard macro vanishes into the mist.

Named macros are slightly more complicated. They have names, just like commands and MLisp functions and can be called by name (or bound to a key). They are defined by using the define-string-macro command (which must be executed by typing **ESC-Xdefine-string-macro** since it isn't usually bound to a key) which asks for the name of the macro and it's body. The body is typed in as a string in the prompt area at the bottom the the screen and hence all special characters in it must be quoted by prefixing them with **↑Q**. A named macro may be executed by typing **ESC-Xname-of-macro** or by binding it to a key with **bind-to-key.**

The current keyboard macro can be converted into a named macro by using the define-keyboard-macro command which takes a name a defines a macro by that name whose body is the current keyboard macro. The current keyboard macro ceases to exist.

13.2. MLisp -- *Mock Lisp*

Unix EMACS contains an interpreter for a language that in many respects resembles Lisp. The primary (some would say only) resemblance between *Mock Lisp* and any real Lisp is the general syntax of a program, which many feel is Lisp's weakest point. The differences include such things as the lack of a **cons** function and a rather peculiar method of passing parameters.

13.2.1. The syntax of MLisp expressions

There are four basic syntactic entities out of which MLisp expressions are built. The two simplest are integer constants (which are optionally signed strings of digits) and string constants (which are sequences of characters bounded by double quote [“"”] characters -- double quotes are included by doubling them: """" is a one character string. The third are names which are used to refer to things: variables or procedures. These three are all tied together by the use of procedure calls. A procedure call is written as a left parenthesis, “(”, a name which refers to the procedure, a list of whitespace separated expressions which serve as arguments, and a closing right parenthesis, “)”. An expression is simply one of these four things: an integer constant, a string constant, a name, or a call which may itself be recursivly composed of other expressions.

String constants may contain the usual **C** excape sequences, "\n" is a newline, "\t" is a tab, "\r" is a carriage return, "\b" is a backspace, "\e" is the escape (033) character, "*nnn*" is the character whose octal representation is *nnn*, and "↑*c*" is the control version of the character *c*.

For example, the following are legal MLisp expressions:

1	The integer constant 1.
"hi"	A two character string constant
"\↑X\↑F"	A two character string constant
"""what?"""	A seven character string constant
(+ 2 2)	An invocation of the "+" function with integer arguments 2 and 2. "+" is the usual addition function. This expression evaluates to the integer 4.
(setq bert (* 4 12))	An invocation of the function *setq* with the variable *bert* as its first argument and and expression that evaluates the product of 4 and 12 as its second argument. The evaluation of this expression assigns the integer 48 to the variable *bert*.
(visit-file "mbox")	An invocation of the function *visit-file* with the string "mbox" as its first argument. Normally the *visit-file* function is tied to the key **↑X↑B**. When it is invoked interactively, either by typing **↑X↑B** or **ESC-Xvisit-file**, it will prompt in the minibuf for the name of the file. When called from MLisp it takes the file name from the parameter list. All of the keyboard-callable function behave this way.

Names may contain virtually any character, except whitespace or parens and they cannot begin with a digit, “"” or “-”.

13.2.2. The evaluation of MLisp expressions

Variables must be declared (*bound*) before they can be used. The declare-global command can be used to declare a global variable; a local is declared by listing it at the beginning of a **progn** or a function body (ie. immediatly after the function name or the word **progn** and before the executable statements). For example:

```
(defun
    (foo i
        (setq i 5)
    )
)
```

defines a rather pointless function called *foo* which declares a single local variable *i* and assigns it the value 5. Unlike real Lisp systems, the list of declared variables is not surrounded by parenthesis.

Expressions evaluate to values that are either integers, strings or markers. Integers and strings are converted automaticly from one to the other type as needed: if a function requires an integer parameter you can pass it a string and the characters that make it up will be parsed as an integer; similarly passing an integer where a string is required will cause the integer to be converted. Variables may have either type and their type is decided dynamically when the assignment is made.

Marker values indicate a position in a buffer. They are not a character number. As insertions and deletions are performed in a buffer, markers automatically follow along, maintaining their position. Only the functions *mark* and *dot* return markers; the user may define ones that do and may assign markers to variables. If a marker is used in a context that requires an integer value then the ordinal of the position within the buffer is used; if a marker is used in a context that requires a string value then the name of the marked buffer is used. For example, if **`there`** has been assigned some marker, then **`(pop-to-buffer there)`** will pop to the marked buffer. **`(goto-character there)`** will set dot to the marked position.

A procedure written in MLisp is simply an expression that is bound to a name. Invoking the name causes the associated expression to be evaluated. Invocation may be triggered either by the evaluation of some expression which calls the procedure, by the user typing it's name to the ESC-X command, or by striking a key to which the procedure name has been bound.

All of the commands listed in section 21 (page 39) may be called as MLisp procedures. Any parameters that they normally prompt the user for are taken as string expressions from the argument list in the same order as they are asked for interactivly. For example, the *switch-to-buffer* command, which is normally tied to the ↑XB key, normally prompts for a buffer name and may be called from MLisp like this: (switch-to-buffer *string-expression*).

13.2.3. Scope issues

There are several sorts of names that may appear in MLisp programs. Procedure, buffer and abbrev table names are all global and occupy distinct name space. For variables there are three cases:

1. Global variables: these variables have a single instance and are created either by using *declare-global*, *set-default* or *setq-default*. Their lifetime is the entire editing session from the time they are created.

2. Local variables: these have an instance for each declaration in a procedure body or local block (*progn*). Their lifetime is the lifetime of the block which declares them. Local declarations nest and hide inner local or global declarations.

3. Buffer-specific variables: these have a default instance and an instance for each buffer in which they have been explicitly given a value. They are created by using *declare-buffer-specific*. When a variable which has been declared to be buffer specific is assigned a value, if an instance for the current buffer hasn't been created then it will be. The value is assigned to the instance associated with the current buffer. If a buffer specific variable is referenced and an instance doesn't exist for this buffer then the default value is used. This default value may be set with either *setq-default* or *set-default*. If a global instance exists when a variable is declared buffer-specific then the global

value becomes the default.

13.2.4. MLisp functions

An MLisp function is defined by executing the *defun* function. For example:

```
(defun
    (silly
        (insert-string "Silly!")
    )
)
```

defines a function called *silly* which, when invoked, just inserts the string "Silly!" into the current buffer.

MLisp has a rather strange (relative to other languages) parameter passing mechanism. The *arg* function, invoked as (arg *i prompt*) evaluates the *i*'th argument of the invoking function if the invoking function was called interactivly or, if the invoking function was not called interactivly, *arg* uses the prompt to ask you for the value. Consider the following function:

```
(defun
    (in-parens
        (insert-string "(")
        (insert-string (arg 1 "String to insert? "))
        (insert-string ")")
    )
)
```

If you type **ESC-Xin-parens** to invoke *in-parens* interactivly then EMACS will ask in the minibuffer "String to insert? " and then insert the string typed into the current buffer surrounded by parenthesis. If *in-parens* is invoked from an MLisp function by **(in-parens "foo")** then the invocation of *arg* inside *in-parens* will evaluate the expression "foo" and the end result will be that the string "(foo)" will be inserted into the buffer.

The function *interactive* may be used to determine whether or not the invoking function was called interactivly. *Nargs* will return the number of arguments passed to the invoking function.

This parameter passing mechanism may be used to do some primitive language extension. For example, if you wanted a statement that executed a statement *n* times, you could use the following:

```
(defun
    (dotimes n
        (setq n (arg 1))
        (while (> n 0)
            (setq n (- n 1))
            (arg 2)
        )
    )
)
```

Given this, the expression **(dotimes 10 (insert-string "<>"))** will insert the string "<>" 10 times. [*Note*: The prompt argument may be omitted if the function can never be called interactivly].

13.2.5. Debugging

Unfortunatly, debugging MLisp functions is something of a black art. The biggest problem right now is that if an MLisp function goes into an infinite loop there is no way to stop it.

There is no breakpoint facility. All that you can do is get a stack trace whenever an error occurs by setting the *stack-trace-on-error* variable. With this set, any time that an error occurs a dump of the MLisp execution call stack and some other information is dumped to the "Stack trace" buffer.

13.3. A Sample MLisp Program

The following piece of MLisp code is the Scribe mode package. Other implementations of EMACS, on ITS and on Multics have *modes* that influence the behaviour of EMACS on a file. This behaviour is usually some sort of language-specific assistance. In Unix EMACS a *mode* is no more that a set of functions, variables and key-bindings. This mode package is designed to be useful when editing Scribe source files.

```
(defun
```

The apply-look function makes the current word "look" different by changing the font that it is printed in. It positions dot at the beginning of the word so you can see where the change will be made and reads a character from the tty. Then it inserts "@c[" (where c *is the character typed) at the front of the word and "]" at the back. Apply-look gets tied to the key* ESC-l *so typing* ESC-l i *when the cursor is positioned on the word "begin" will change the word to "@i[begin]".*

```
    (apply-look go-forward
        (save-excursion c
            (if (! (eolp)) (forward-character))
            (setq go-forward -1)
            (backward-word)
            (setq c (get-tty-character))
            (if (> c ' ')
                (progn (insert-character '@')
                    (insert-character c)
                    (insert-character '[')
                    (forward-word)
                    (setq go-forward (dot))
                    (insert-character ']')
                )
            )
        )
        (if (= go-forward (dot)) (forward-character))
    )
)
```

```
(defun
                                        This function is called to set a buffer into Scribe mode
    (scribe-mode
        (remove-all-local-bindings)
                                        If the string "LastEditDate= """ exists in the first 2000 characters of the document then the
                                        following string constant is changed to the current date. The intent of this is that you should stick
                                        at the beginning of your file a line like: "@string(LastEditDate= "Sat Jul 11 17:59:01 1981")".
                                        This will automatically get changed each time you edit the file to reflect that last date on which
                                        the file was edited.
        (if (! buffer-is-modified)
            (save-excursion
                (error-occurred
                    (goto-character 2000)
                    (search-reverse "LastEditDate=""")
                    (search-forward """")
                    (set-mark)
                    (search-forward """")
                    (backward-character)
                    (delete-to-killbuffer)
                    (insert-string (current-time))
                    (setq buffer-is-modified 0)
                )
            )
        )
        (local-bind-to-key "justify-paragraph" "\ej")
        (local-bind-to-key "apply-look" "\el")
        (setq right-margin 77)
        (setq mode-string "Scribe")
        (setq case-fold-search 1)
        (use-syntax-table "text-mode")
        (modify-syntax-entry "w    -'")
        (use-abbrev-table "text-mode")
        (setq left-margin 1)
        (novalue)
    )
)

(novalue)
```

13.4. More on Invoking EMACS

When EMACS is invoked, it does several things that are not of too much interest to the beginning user.

1. EMACS looks for a file called "**.emacs_pro**" in your home directory, if it exists then it is loaded, with the *load* command. This is the mechanism used for user profiles -- in your **.emacs_pro** file, place the commands needed to customize EMACS to suit your taste. If a user has not set up an **.emacs_pro** file then EMACS will use a site-specific default file for initialization. At CMU this file is named /usr/local/lib/emacs/maclib/profile.ml

2. EMACS will then interprete its command line switches. "-l<filename>" loads the given file (only one may be named), "-e<funcname>" executes the named function (again, only one may be named). -l and -e are executed in that order, after the user profile is read, but before and file visits are done. This is intended to be used along with the csh alias mechanism to allow you to invoke EMACS packages from the shell (that is, assuming that there is anyone out there who still uses the shell for anything other than to run under EMACS!). For example: "**alias rmail emacs -lrmail -ermail-com**" will cause the csh "rmail" command to invoke EMACS running rmail. Exiting rmail will exit EMACS.

3. If neither *argv* nor *argc* have yet been called (eg. by your startup or by the command line named package) then the list of arguments will be considered as file names and will be visited; if there are no arguments then the arguments passed to the last invocation of EMACS will be used.

4. Finally, EMACS invokes it's keyboard command interpreter, and eventually terminates.

14. Searching

EMACS is capable of performing two kinds of searches[1]. There are two parallel sets of searching and replacement commands that differ only in the kind of search performed.

14.1. Simple searches

The commands *search-forward*, *search-reverse*, *query-replace-string* and *replace-string* all do simple searches. That is, the search string that they use is matched directly against successive substrings of the buffer. The characters of the search string have no special meaning. These search forms are the easiest to understand and are what most people will want to use. They are what is conventionally bound to ↑S, ↑R, ESC-Q and ESC-R.

14.2. Regular Expression searches

The commands *re-search-forward*, *re-search-reverse*, *re-query-replace-string*, *re-replace-string* and *looking-at* all do regular expression searches. The search string is interpreted as a regular expression and matched against the buffer according to the following rules:

1. Any character except a special character matches itself. Special characters are '\' '[' '.' and sometimes '↑' '*' '$'.

2. A '.' matches any character except newline.

3. A '\' followed by any character except those mentioned in the following rules matches that character.

4. A '\w' Matches any word character, as defined by the syntax tables.

5. A '\W' Matches any non-word character, as defined by the syntax tables.

6. A '\b' Matches at a boundary between a word and a non-word character, as defined by the syntax tables.

7. A '\B' Matches anywhere but at a boundary between a word and a non-word character, as defined by the syntax tables.

8. A '\'' Matches at the beginning of the buffer.

[1] *Regular* and *Vanilla* for those of you with no taste

9. A '\'' Matches at the end of the buffer.

10. A '\<' Matches anywhere before dot.

11. A '\>' Matches anywhere after dot.

12. A '\=' Matches at dot.

13. A nonempty string *s* bracketed "[*s*]" (or "[↑ *s*]" matches any character in (or not in) *s*. In *s*, '\' has no special meaning, and ']' may only appear as the first letter. A substring *a-b*, with *a* and *b* in ascending ASCII order, stands for the inclusive range of ASCII characters.

14. A '\' followed by a digit *n* matches a copy of the string that the bracketed regular expression beginning with the *n* th '\(' matched.

15. A regular expression of one of the preceeding forms followed by '*' matches a sequence of 0 or more matches of the regular expression.

16. A regular expression, *x*, bracketed "\(*x* \)" matches what *x* matches.

17. A regular expression of this or one of the preceeding forms, *x*, followed by a regular expression of one of the preceeding forms, *y* matches a match for *x* followed by a match for *y*, with the *x* match being as long as possible while still permitting a *y* match.

18. A regular expression of one of the preceeding forms preceded by '↑' (or followed by '$'), is constrained to matches that begin at the left (or end at the right) end of a line.

19. A sequence of regular expressions of one of the preceeding forms seperated by '\|'s matches any one of the regular expressions.

20. A regular expression of one of the preceeding forms picks out the longest amongst the leftmost matches if searching forward, rightmost if searching backward.

21. An empty regular expression stands for a copy of the last regular expression encountered.

In addition, in the replacement commands, *re-query-replace-string* and *re-replace-string*, the characters in the replacement string are specially interpreted:

- Any character except a special character is inserted unchanged.

- A '\' followed by any character except a digit causes that character to be inserted unchanged.

- A '\' followed by a digit *n* causes the string matched by the *n*th bracketed expression to be inserted.

- An '&' causes the string matched by the entire search string to be inserted.

The following examples should clear a little of the mud:

`Pika`	Matches the simple string "Pika".

Whiskey.*Jack Matches the string "Whiskey", followed by the longest possible sequence of non-newline characters, followed by the string "Jack". Think of it as finding the first line that contains the string "Whiskey" followed eventually on the same line by the string "Jack"

[a-z][a-z]* Matches a non-null sequence of lower case alphabetics. Using this in the *re-replace-string* command along with the replacement string "**(&)**" will place parenthesis around all sequences of lower case alphabetics.

Guiness\|Bass Matches either the string 'Guiness' or the string 'Bass'.

\Bed\b Matches 'ed' found as the suffix of a word.

\bsilly\W*twit\b
Matches the sequence of words 'silly' and 'twit' seperated by arbitrary punctuation.

15. Keymaps

When a user is typing to EMACS the keystrokes are interpreted using a *keymap.* A keymap is just a table with one entry for each character in the ASCII character set. Each entry either names a function or another keymap. When the user strikes a key, the corresponding keymap entry is examined and the indicated action is performed. If the key is bound to a function, then that function will be invoked. If the key is bound to another keymap then that keymap is used for interpreting the next keystroke.

There is always a global keymap and a local keymap, as keys are read from the keyboard the two trees are traversed in parallel (you can think of keymaps as FSMs, with keystrokes triggering transitions). When either of the traversals reaches a leaf, that function is invoked and interpretation is reset to the roots of the trees.

The root keymaps are selected using the *use-global-map* or *use-local-map* commands. A new empty keymap is created using the *define-keymap* command.

The contents of a keymap can be changed by using the *bind-to-key* and *local-bind-to-key* commands. These two commands take two arguments: the name of the function to be bound and the keystroke sequence to which it is to be bound. This keystroke sequence is interpreted relative to the current local or global keymaps. For example, **(bind-to-key "define-keymap" "\↑Zd")** binds the *define-keymap* function to the keystroke sequence '↑Z' followed by '**d**'.

A named keymap behaves just like a function, it can be bound to a key or executed within an MLisp function. When it is executed from within an MLisp function, it causes the next keystroke to be interpreted relative to that map.

The following sample uses the keymap to partially simulate the *vi* editor. Different keymaps are used to simulate the different modes in *vi*: command mode and insertion mode.

```
(defun
    (insert-before              ; Enter insertion mode
        (use-global-map "vi-insertion-mode"))

    (insert-after               ; Also enter insertion mode, but after
                                ; the current character
        (forward-character)
        (use-global-map "vi-insertion-mode"))
    (exit-insertion                 ; Exit insertion mode and return to
                                    ; command mode
        (use-global-map "vi-command-mode"))

    (replace-one
        (insert-character (get-tty-character))
        (delete-next-character))

    (next-skip
        (beginning-of-line)
        (next-line)
        (skip-white-space))

    (prev-skip
        (beginning-of-line)
        (previous-line)
        (skip-white-space))

    (skip-white-space
        (while (& (! (eolp)) (| (= (following-char) ' ') (= (following-char) '↑i')))
            (forward-character)))

    (vi                             ; Start behaving like vi
        (use-global-map "vi-command-mode"))
)

; setup vi mode tables
(define-keymap "vi-command-mode")
(define-keymap "vi-insertion-mode")

(use-global-map "vi-insertion-mode"); Setup the insertion mode map
(bind-to-key "execute-extended-command" "\↑X")
(progn i
    (setq i ' ')
    (while (< i 0177)
        (bind-to-key "self-insert" i)
        (setq i (+ i 1))))
(bind-to-key "self-insert" "\011")
(bind-to-key "newline" "\015")
(bind-to-key "self-insert" "\012")
(bind-to-key "delete-previous-character" "\010")
(bind-to-key "delete-previous-character" "\177")
(bind-to-key "exit-insertion" "\033")

(use-global-map "vi-command-mode"); Setup the command mode map
(bind-to-key "execute-extended-command" "\↑X")
(bind-to-key "next-line" "\↑n")
(bind-to-key "previous-line" "\↑p")
(bind-to-key "forward-word" "w")
(bind-to-key "backward-word" "b")
(bind-to-key "search-forward" "/")
(bind-to-key "search-reverse" "?")
(bind-to-key "beginning-of-line" "0")
(bind-to-key "end-of-line" "$")
(bind-to-key "forward-character" " ")
(bind-to-key "backward-character" "\↑h")
(bind-to-key "backward-character" "h")
(bind-to-key "insert-after" "a")
(bind-to-key "insert-before" "i")
(bind-to-key "replace-one" "r")
(bind-to-key "next-skip" "+")
(bind-to-key "next-skip" "\↑m")
(bind-to-key "prev-skip" "-")
(use-global-map "default-global-keymap")
```

16. Region Restrictions

The portion of the buffer which EMACS considers visible when it performs editing operations may be restricted to some subregion of the whole buffer.

The *narrow-region* command sets the restriction to encompass the region between dot and mark. Text outside this region will henceforth be totally invisible. It won't appear on the screen and it won't be manipulable by any editing commands. It will, however, be read and written by file manipulation commands like *read-file* and *write-current-file.* This can be useful, for instance, when you want to perform a replacement within a few paragraphs: just narrow down to a region enclosing the paragraphs and execute *replace-string.*

The *widen-region* command sets the restriction to encompass the entire buffer. It is usually used after a *narrow-region* to restore EMACS's attention to the whole buffer.

Save-restriction is only useful to people writing MLisp programs. It is used to save the region restriction for the current buffer (and ***only*** the region restriction) during the execution of some subexpression that presumably uses region restrictions. The value of **`(save-restriction expressions...)`** is the value of the last expression evaluated.

17. Mode Lines

A *mode line* is the line of descriptive text that appears just below a window on the screen. It usually provides a description of the state of the buffer and is usually shown in reverse video. The standard mode line shows the name of the buffer, an '*' if the buffer has been modified, the name of the file associated with the buffer, the *mode* of the buffer, the current position of dot within the buffer expressed as a percentage of the buffer size and and indication of the nesting within *recursive-edit*'s which is shown by wrapping the mode line in an appropriate number of '[' ']' pairs.

It is often the case that for some silly or practical reason one wants to alter the layout of the mode line, to show more, less or different information. EMACS has a fairly general facility for doing this. Each buffer has associated with it a format string that describes the layout of the mode line for that buffer whenever it appears in a window. The format string is interpreted in a manner much like the format argument to the C printf subroutine. Unadorned characters appear in the mode line unchanged. The '%' character and the following format designator character cause some special string to appear in the mode line in their place. The format designators are:

b	Inserts the name of the buffer.
f	Inserts the name of the file associated with the buffer.
m	Inserts the value of the buffer-specific variable *mode-string.*
M	Inserts the value of the variable *global-mode-string.*
p	Inserts the position of "dot" as a percentage.
*****	Inserts an '*' if the buffer has been modified.
[	Inserts (recursion-depth) '['s.
]	Inserts (recursion-depth) ']'s.

If a number *n* appears between the '%' and the format designator then the inserted string is constrained to be exactly *n* characters wide. Either by padding or truncating on the right.

At CMU the default mode line is built using the following format:

```
" %[Buffer: %b%*  File: %f  %M (%m)  %p%]"
```

The following variables are involved in generating mode lines:

mode-line-format This is the buffer specific variable that provides the format of a buffers mode line.

default-mode-line-format
This is the value to which *mode-line-format* is initialized when a buffer is created.

mode-string This buffer-specific string variable can be inserted into the mode line by using '%m' in the format. This is it's only use by EMACS. Usually, mode packages (like 'lisp-mode' or 'c-mode') put some string into *mode-string* to indicate the mode of the buffer. It is the appearance of this piece of descriptive information that gives the mode line its name.

*global-mode-string*This is similar to *mode-string* except that it is global -- the same string will be inserted into all mode lines by '%M'. It is usually used for information of global interest. For example, the time package puts the current time of day and load average there.

18. Multiple Processes under EMACS

EMACS has the ability to handle multiple interactive subprocesses. The following is a sketchy description of this capability.

In general, you will *not* want to use any of the functions described in the rest of this section. Instead, you should be using one of the supplied packages that invoke them, see 20.14 page 32. For example, the "shell" command provides you with a window into an interactive shell and the "time" package puts the current time and load average (continuously updated) into the mode line.

Multiple interactive processes can be started under EMACS (using "start-process" or "start-filtered-process"). Processes are tied to a buffer at inception and are thereafter known by this buffer name. Input can be sent to a process from the region or a string, and output from processes is normally attached to the end of the process buffer. There is also the ability to have EMACS call an arbitrary MLISP procedure to process the output each time it arrives from a process (see "start-filtered-process").

Many of the procedures dealing with process management use the concept of "current-process" and "active-process". The current-process is usually the most recent process to have been started. Two events can cause the current-process to change:

1. When the present current-process dies, the most recent of the remaining processes is popped up to take its place.

2. The current-process can be explicitly changed using the "change-current-process" command.

The active-process refers to the current-process, unless the current buffer is a live process in which case it refers to the current buffer.

Below is list of the current mlisp procedures for using processes:

active-process **[unbound]**: (active-process) -- Returns the name of the active process as defined in the section describing the process mechanism.

change-current-process **[unbound]**: (change-current-process "process-name") -- Sets the current process to the one named.

continue-process **[unbound]**: (continue-process "process-name") -- Continue a process stopped by *stop-process.*

current-process **[unbound]**: (current-process) -- Returns the name of the current process as defined in the section describing the process mechanism.

eot-process **[unbound]**: (eot-process "process-name") -- Send an EOT to the process.

int-process **[unbound]**: (int-process "process-name") -- Send an interrupt signal to the process.

kill-process **[unbound]**: (kill-process "process-name") -- Send a kill signal to the process.

list-processes **[unbound]**: (list-processes) -- Analagous to "list-buffers". Processes which have died only appear once in this list before completely disappearing.

process-filter-name **[unbound]**: Returns the name of the filter procedure attached to some buffer.

process-id **[unbound]**: Returns the process id of the process attached to some buffer.

process-output **[unbound]**: (process-output) -- Can only be called by the *on-output-procedure* to procure the output generated by the process whose name is given by *MPX-process.* Returns the output as a string.

process-status **[unbound]**: (process-status "process-name") -- Returns -1 if "process-name" isn't a process, 0 if the process is stopped, and 1 if the process is running.

quit-process **[unbound]**: (quit-process "process-name") -- Send a quit signal to the process.

region-to-process **[unbound]**: (region-to-process "process-name") -- The region is wrapped up and sent to the process.

Variable *silently-kill-processes*: If ON EMACS will kill processes when it exits *without* asking any questions. Normally, if you have processes running when EMACS exits, the question "You have processes on the prowl, should I hunt them down for you" is asked. (default OFF)

start-filtered-process **[unbound]**: (start-filtered-process "command" "buffer-name" "on-output-procedure") -- Does the same thing as start-process except that things are set up so that "on-output-procedure" is automatically called whenever output has been received from this process. This procedure can access the name of the process producing the output by refering to the variable *MPX-process*, and can retrieve the output itself by calling the procedure *process-output*.

> **The filter procedure must be careful to avoid generating side-effects (eg. *search-forward*). Moreover, if it attempts to go to the terminal for information, output from other processes may be lost.**

start-process **[unbound]**: (start-process "command" "buffer-name") -- The home shell is used to start a process executing the command. This process is tied to the buffer "buffer-name" unless it is null in which case the "Command execution" buffer is used. Output from the process is automatically attached to the end of the buffer. Each time this is done, the mark is left at the end of the output (which is the end of the buffer).

stop-process **[unbound]**: (stop-process "process-name") -- Tell the process to stop by sending it a stop signal. Use *continue-process* to carry on.

string-to-process **[unbound]**: (string-to-process "process-name" "string") -- The string is sent to the process.

18.1. Blocking

When too many characters are sent to a process in one gulp, the send will be blocked until the process has removed sufficient characters from the buffer. The send will then be automatically continued. Normally this process is invisible to the EMACS user, but if the process has been stopped, the send will not be unblocked and further attempts to send to the process will result in an overwrite error message.

18.2. Buffer Truncation

EMACS does not allow process buffers to grow without bound. When a process buffer exceeds the value of the variable *process-buffer-size*, 500 characters are erased from the beginning of the buffer. The default value for *process-buffer-size* is 10,000.

18.3. Problems

The most obvious problem with allowing multiple interactive processes is that it is too easy to start up useless jobs which drag everyone down. Also when checkpointing is done, all buffers including the process buffers are checkpointed. So if you have a one line buffer keeping time, it will take more system time to checkpoint it than it will to keep it updated once a minute.

In addition to anti-social problems, there are some real bugs remaining:

- Sometimes when starting a process, it will inexplicably expire immediately. This often happens to the first process you fire up.

- Subprocesses are assumed to not want to try fancy things with the terminal. EMACS doesn't know how to handle this and for now more or less ignores stty requests from processes. This means that csh cannot be used from within EMACS. Running chat and ftp can also cause problems. Someday, EMACS should try to handle stty's.

- The worst problem is that background processes started outside EMACS will cause EMACS to hang when they finally finish. This might get fixed if I want to think about it.

- If EMACS does crash or hang, you will find several orphan processes left hanging around. It is best to do a ps and get rid of them.

19. The EMACS database facility

Unix EMACS provides a set of commands for dealing with databases of a rather primitive form. These databases are intended to be used in *help* facilities to find documentation for a given keyword, but they have many other uses: managed mailboxes or nodes in an *info* tree.

A *database* is a set of (key, content) pairs which may be retrieved or stored based on the key. Both the key and the content may be arbitrary strings of characters. The content may be long, but there are restrictions on the aggragate length of the keys.

A *database search list* is a list of databases. When a key is looked up in a database search list the databases in the search list are examined in order for one containing the key. The content corresponding to the first key that matches is returned. When a key is to have its content changed only the first database in the search list is used.

The commands available for dealing with databases are:

extend-database-search-list **[unbound]**: (extend-database-search-list dbname filename) adds the given data base file to the data base search list (dbname). If the database is already in the search list then it is left, otherwise the new database is added at the beginning of the list of databases.

fetch-database-entry **[unbound]**: (fetch-database-entry dbname key) takes the entry in the data base corresponding to the given key and inserts it into the current buffer.

list-databases **[unbound]**: (list-databases) lists all data base search lists.

put-database-entry **[unbound]**: (put-database-entry dbname key) takes the current buffer and stores it into the named database under the given key.

There are four Unix commands provided for dealing with EMACS data bases (these are commands that you give to the shell, not EMACS):

1. **dbadd** -- add entry to an Emacs data base
   ```
   dbadd dbname key
   ```

2. **dbcreate** -- create an Emacs data base
   ```
   dbcreate dbname
   ```

3. **dblist** -- list contents of an Emacs data base
   ```
   dblist dbname [ -l ] [ -p ] newdbname
   ```

4. **dbprint** -- print an entry from an Emacs data base
   ```
   dbprint dbname key
   ```

Dbadd adds the text from the standard input to the named database using the given key. **Dbcreate** creates the named database, making it empty. **Dbprint** prints the contents of the entry from the database with the given key.

Dblist with no arguments simply lists the keys of all the items in the database. With the **-l** option it prints some internal information from the database of no interest to anyone but the implementor. The **-p** option causes the key and content of every entry to be listed as a shell command file which when executed will repeatedly invoke **dbadd** to rebuild the database. This form of **dblist** is handy when you want a readable ascii file representation of a data base for shipping around or editing. Databases should be recreated periodically to garbage collect them.

20. Packages

This chapter contains a description of a few of the packages that have been written for EMACS in MLisp. To load some package, just type "ESC-X load *PackageName*". The title of each following section contains the name of the package before the '--'.

20.1. abbrev -- define abbreviation for word in buffer

abbreviate-word	Prompts for an abbreviation for the current word. If a prefix argument is provided, the specified number of words are taken as the "word" to abbreviate. In any case, the minibuffer will show exactly what is being abbreviated.

20.2. buff -- one-line buffer list

Loading the *buff* package replaces the binding for ↑X-↑B (usually *list-buffers*) with *one-line-buffer-list.*

one-line-buffer-list Gives a one-line buffer list in the mini-buffer. If the buffer list is longer than one line, it will print a line at a time and wait for a character to be typed before moving to the next line. Buffers that have been changed since they were last saved are prefixed with an asterisk (*), buffers with no associated file are prefixed with a hash-mark (#), and empty buffers are prefixed with an at-sign (@).

20.3. Buffer-edit -- a buffer management function

This package provides a very nice buffer management package intended to replace the *list-buffers* function normally bound to ↑X↑B. It pops up a window that contains a buffer listing, and lets you move around that buffer listing marking buffers for deletion, saving, unsaving, reverting, and so forth. This ability is incredibly useful when you are editing a big system that has all kinds of files all over the place, because it lets you move around freely from one file to another without having to remember or type buffer names.

When *buffer-edit* is run, normally by typing ↑X↑B, it pops up a window whose contents is a buffer listing, sorted so that the file buffers come first, with the cursor positioned on the line corresponding to the buffer in which the command was executed. You can move the cursor from line to line in that buffer listing by using the ordinary cursor-movement commands, or by using "n" for next and "p" for previous. When the cursor is positioned on a line corresponding to some buffer *B*, various commands can be typed that will change the disposition of *B* when the buffer is exited:

d Delete the buffer. The buffer will be removed from the editor's tables using the EMACS *delete-buffer* command. This command will not write out a modified buffer, so that information will be lost if you delete a buffer with **d**.

c Close the buffer. The buffer will be written out to its attached file (if there is one) and then it will be deleted as with the **d** command.

r Revert the buffer. If it is not a file buffer, nothing will happen. If it is a file buffer, then the current contents of the file will be read into the buffer, replacing its current contents. If you have made extensive modifications to a buffer and then decide that you want to start over again from the file copy, you can use this command. The buffer will not be deleted from EMACS' list of buffers.

s Save the buffer. The buffer will be written out to its attached file, exactly as with the ↑X↑S command.

m Mark the buffer as unmodified. This will remove the "M" flag from the buffer listing, and mark the buffer as not in need of being saved, but will not actually write the buffer out to any file.

u Unmark the buffer. Any action flag set by one of the above commands will be removed.

In addition to those commands shown above that "mark" a buffer for processing when the buffer menu is exited, there are commands that have immediate action at the instant that they are typed:

e	Begin a recursive edit on the contents of the buffer.
q	Exit without processing. If you change your mind about all of the buffer operations you have flagged, you can go back and unmark them all by typing a **u** command for each one, or you can just type a *q* command, which will get you out of the buffer edit back to where you entered it from.
g	Exit and go to a buffer. This is the normal way of exiting from buffer-edit: you find the line corresponding to the buffer that you would like to edit next, and type g. All of the marks are processed, the various *save*, *delete*, and *revert* operations are performed, and the screen is filled with the contents of the indicated buffer.
?	Print some help text that includes a summary of these commands.

20.4. c-mode -- simple assist for C programs

begin-C-comment	(ESC-') Initiates the typing in of a comment. Moves the cursor over to the comment column, inserts "/* " and turns on autofill. If ESC-' is typed in the first column, the the comment begins there, otherwise it begins where ever *comment-column* says it should.
end-C-comment	(ESC-') Closes off the current comment.
indent-C-procedure	(ESC-j) Takes the current function (the one in which dot is) and fixes up its indentation by running it through the "indent" program.

20.5. capword -- different behavior for word capitalizations

The built-in EMACS functions *case-word-upper*, *case-word-lower*, and *case-word-capitalize* all leave the cursor where it began, and perform their operation on the word containing the cursor. Many people prefer to have these functions skip forward over a word after capitalizing or uncapitalizing it. These functions provide that service.

The *capword* package defines three functions, *upper-case-word*, *lower-case-word*, and *capitalize-word*. Normally they are bound to ESC-U, ESC-L, and *ESC-C* respectively, though this package does not set up those bindings.

20.6. dired -- directory editor

The *dired* package implements the *dired* command which provides some simple convenient directory editing facilities. When you run *dired* it will ask for the name of a directory, displays a listing of it in a buffer, and processes commands to examine files and possibly mark them for deletion. When you're through with *dired* it actually deletes the marked files, after asking for confirmation. The commands it recognizes are:

d	Marks the current file for deletion. A 'D' will appear at the left margin. It does not actually delete the file, it just marks it. The deletion will be performed when *dired* is exited. It also makes the next file be the current one.

u	Removes the deletion mark from the current file. This is the command to use if you change your mind about deleting a file. It also makes the next file be the current one.
RUBOUT	Removes the deletion mark from the line preceeding the current one. If you mark a file for deletion with 'd' the current file will be advanced to the next line. RUBOUT undoes both the advancing and the marking for deletion.
e, v	Examine a file put putting it in another window and doing a recursive-edit on it. To resume *dired* type ↑C.
r	Removes the current file from the directory listing. It doesn't delete the file, it just gets rid of the directory listing entry. Use it to remove some of the clutter on your screen.
q, ↑C	Exits *dired.* For each file that has been marked for deletion you will be asked for confirmation. If you answer 'y' the file will be deleted, otherwise not.
n, ↑N	Moves to the next entry in the directory listing.
p, ↑P	Moves to the previous entry in the directory listing.
↑V	Moves to the next page in the directory listing.
ESC-v	Moves to the previous page in the directory listing.
ESC-<	Moves to the beginning of the directory listing.
ESC->	Moves to the end of the directory listing.

20.7. goto -- go to position in buffer

goto-line	Moves the cursor to beginning of the indicated line. The line number is taken from the prefix argument if it is provided, it is prompted for otherwise. Line numbering starts at 1.
goto-percent	Moves dot to the indicated percentage of the buffer. The percentage is taken from the prefix argument if it is provided, it is prompted for otherwise. **(goto-percent *n*)** goes to the character that is *n*% from the beginning of the buffer.

20.8. incr-search -- ITS style incremental search

ITS EMACS has a the search command that is unusual in that it is "incremental"; it begins to search before you have finished typing the search string. As you type in the search string, EMACS shows you where it would be found. When you have typed enough characters to identify the place you want, you can stop. The incr-search package perfectly emulates this in Unix EMACS. Typically one binds *incremental-search* to ↑S and *reverse-incremental-search* to ↑**R.**

The command to search is ↑S (incremental-search). ↑S reads in characters and positions the cursor at the first occurrence of the characters that you have typed. If you type ↑S and then F, the cursor moves right after the first "F". Type an "O", and see the cursor move to after the first "FO". After another "O", the cursor is after the first "FOO" after the place where you started the search. At the same time, the "FOO" has echoed at the bottom of the screen.

If you type a mistaken character, you can rub it out. After the FOO, typing a rubout makes the "O" disappear from the bottom of the screen, leaving only "FO". The cursor moves back to the "FO". Rubbing out the "O" and "F" moves the cursor back to where you started the search.

When you are satisfied with the place you have reached, you can type an ESC, which stops searching, leaving the cursor where the search brought it. Also, any command not specially meaningful in searches stops the searching and is then executed. Thus, typing ↑A would exit the search and then move to the beginning of the line. ESC is necessary only if the next command you want to type is a printing character, Rubout, ESC or another search command, since those are the characters that would not exit the search.

Sometimes you search for "FOO" and find it, but not the one you expected to find. There was a second FOO that you forgot about, before the one you were looking for. Then type another ↑S and the cursor will find the next FOO. This can be done any number of times. If you overshoot, you can rub out the ↑S's. You can also repeat the search after exiting it, if the first thing you type after entering another search (when the argument is still empty) is a ↑S.

If your string is not found at all, the echo area says "Failing I-Search". The cursor is after the place where EMACS found as much of your string as it could. Thus, if you search for FOOT, and there is no FOOT, you might see the cursor after the FOO in FOOL. At this point there are several things you can do. If your string was mistyped, you can rub some of it out and correct it. If you like the place you have found, you can type ESC or some other EMACS command to "accept what the search offered". Or you can type ↑G, which throws away the characters that could not be found (the "T" in "FOOT"), leaving those that were found (the "FOO" in "FOOT"). A second ↑G at that point undoes the search entirely.

The ↑G "quit" command does special things during searches; just what, depends on the status of the search. If the search has found what you specified and is waiting for input, ↑G cancels the entire search. The cursor moves back to where you started the search. If ↑G is typed while the search is actually searching for something or updating the display, or after search failed to find some of your input (having searched all the way to the end of the file), then only the characters which have not been found are discarded. Having discarded them, the search is now successful and waiting for more input, so a second ↑G will cancel the entire search. Make sure you wait for the first ↑G to ding the bell before typing the second one; if typed too soon, the second ↑G may be confused with the first and effectively lost.

You can also type ↑R at any time to start searching backwards. If a search fails because the place you started was too late in the file, you should do this. Repeated ↑R's keep looking for more occurrences backwards. A ↑S starts going forwards again. ↑R's can be rubbed out just like anything else. If you know that you want to search backwards, you can use ↑R instead of ↑S to start the search, because ↑R is also a command (reverse-incremental-search) to search backward.

20.9. ind-region -- indent (slide) blocks of lines left or right

The *ind-region* package provides a function that will move a block of text lines left or right, for manually meddling with indentation. The set of lines that it operates on is defined by point and mark, but in order to behave intuitively it doesn't quite use point and mark as a region. In particular, it will include the complete contents of any line if any character of that line falls in the marked region, and it will also include a line if the first character of that line is right after the end of the region. This behavior, while it sounds unusual, provides visual fidelity: if you set the mark anywhere on one line, and then move the point to anywhere on another line (including their beginnings or ends, respectively), then those lines will be included in the set of lines that is indented left or right.

If no argument is provided, the function will assume an indentation of +4, which is a right shift of 4 spaces. In all cases, after the function has finished indenting a line it will compute the minimal sequence of tabs and spaces to effect the indentation.

20.10. info -- documentation reader

Info is a system which lets you browse through the documentation for various systems. In particular, all the EMACS documentation is available online through it. Both the *describe-command* and *describe-variable* functions use it.

Rather than document *Info* extensivly here, I suggest that you run *Info* and use it to describe itself.

20.11. killring -- fancy text killing package

This package defines commands for killing and unkilling text. Commands to delete words. lines, and regions actually send the text to a ring of killbuffers, where they can be yanked back. Multiple killing commands in succession will concatenate text to the same buffer, so a single unkill can bring it all back. The unkill-pop command can cycle the kill ring to retrieve previously-killed stuff.

The following keys are redefined:

- **↑W**
  kill-region
- **ESC-w**
  copy-region
- **↑K**
  kill-lines
- **ESC-k**
  copy-lines
- **ESC-d**
  kill-word
- **ESC-h**
  backward-kill-word
- **ESC-del**
  backward-kill-word
- **ESC-a**

append next kill (pretend previous command was ↑K)

- **↑Y**
  unkill

- **ESC-y**
  unkill-pop (kill the region, back up one on the kill ring, and unkill)

There are usually four buffers in the killring. If you want more buffers in the ring (say 8), execute the following mlisp functions BEFORE you load this file:

```
(setq-default nrings 8)
```

The ↑K function will behave pretty much the same as the old *delete-to-end-of-line* did, unless you want something better. The improved version bases its behavior on the horizontal position of the cursor at the time the command is issued. If the cursor is at the beginning of the line, the command will assume you want to kill the entire line, including the return at the end. If you're at the end of the line, then it will remove the return separating this line from the next. Otherwise, it will kill just to the end of the line. To get this function, execute the following mlisp functions *after* you load this file:

```
(setq-default &kill-lines-magic 1)
```

20.12. mhe -- a mail management system based on MH

Mhe is an Emacs-based system that is used as a visual front end to the MH mail system. MH is the Rand Mail Handler, which is available under license from the Rand Corporation. Mhe is used as a mail program to send, receive, classify, move, archive, search, and edit mail using the basic MH programs as the underlying mechanism. While mhe can certainly be loaded from any instance of EMACS, the customary usage is to use mhe for a login shell, or else to execute it immediately after login, and then to sit in it all day, using it as both an editor and a mail reader.

When initially run, mhe presents you with a buffer containing a listing of the headers of the mail messages in your current mail folder; you can then peruse this buffer with all of the usual EMACS motion and search commands. To delete a message, you position the cursor on the line corresponding to that message and type "D"; to reply to a message, you position the cursor on the line corresponding to it and type "R". All of the basic mail-handling commands in mhe are single-character commands, as follows:

n	move cursor to next line
p	move cursor to previous line
t	type this message (the message represented by the current line). Pops up a window and shows the message in it. Mhe key bindings are still in effect while the cursor is in that window.
d	delete this message. Marks it with a "D", and arranges for it to be deleted when the mhe session is terminated.
↑	move this message to another folder. Prompts for its name. Marks it with a "↑" and arranges for it to be moved with the mhe session is terminated.
!	repeat previous ↑ (move) command. Uses same destination folder as previous command, so no prompting is done.
u	undelete/unmove: cancel delete or move command for this message. Since the deleting and moving are not performed until mhe exits, those commands can be undone.

m	mail a message. Pops up a window whose contents are an empty mail message; you fill in the "To:", "Subject:", and "Cc:" fields as you wish. You can add "Fcc:" fields for file copies, "Bcc:" fields for blind copies, and any other fields that you wish (such as "Reply-to:", etc.). Your standard EMACS key bindings will be used in this window. When you exit from the recursive edit with ↑X↑C, you will be asked for instructions on handling the message, e.g. quit, send it, go back and edit it some more.
r	reply to the current message. Splits the screen, showing the message text in one window and the reply in the other. Quite similar to the "mail" command, except that the "Subject:", "To:", and "Cc:" fields are filled in for you. You can change them if you want, of course. When you send the reply, the original message will be annotated with a "Replied:" field and the date, and the letter "R" will appear in the header listing.
f	forward the current message. Pops up a message composition window, just like the "m" command, except that its initial contents are the contents of the current message. When you send the message, the original that you forwarded will be marked with an annotation showing that it has been forwarded to someone, and the letter "R" will appear in the header listing.
e	edit the current message. This command works just like the "type" command described above, except that the keyboard has its "edit" key bindings, so that you can change the message if you want.
i	incorporate new mail. If the banner line shows that you have received new mail, you can fetch it with this command. If you are currently working in some folder besides +inbox, and if there is mail, then mhe will switch to folder +inbox before incorporating the mail.
g	get a new mail folder. Prompts you for the name of a new folder, and then creates a new header buffer in the name of that folder. The old header buffer is not destroyed, so that you can switch back and forth between them as you see fit.
b	get a bboard (bulletin board, otherwise known as newsgroup) folder. Mhe lets you read newsgroup directories just as if they were mail in a mail folder.
↑X↑C	Exit from Mhe.
?	Pop up a help window. Its topmost few lines give a command summary, and if you scroll it down, various further instructions are given.

Whenever the cursor is positioned in a header buffer, the above-mentioned key bindings are in effect. In addition, all of the ↑X-prefix key bindings from your profile are left untouched, as are various other standard EMACS key bindings like ESC-, ↑S, and so forth.

20.13. occur -- find occurances of a string

The *occur* package allows one to find the occurances of a string in a buffer. It contains one function

Occurances When invoked, prompts with "Search for all occurances of: ". It then lists (in a new buffer) all lines contain the string you type following *dot*. Possible options (listed at the bottom of the screen) allow you to page through the listing buffer or abort the function.

In addition, a global variable controls the action of the function:

&Occurances-Extra-Lines
is a global variable that controls how many extra surrounding lines are printed in addition to the line containing the string found. If this variable is 0 then NO additional lines are printed. If this variable is greater than 0 then it will print that many lines above and below the line on which the string was found. When printing more than one line per match in

this fashion, it will also print a seperator of '---------------' so you can tell where the different matches begin and end. At the end of the buffer it prints '<<<End of Occur>>>'.

20.14. process -- high level process manipulation

The process package provides high level access to the process control features of Unix EMACS. It allows you to interact with a shell through an EMACS window, just as though you were talking to the shell normally.

shell	The *shell* command is used to either start or reenter a shell process. When the shell command is executed, if a shell process doesn't exist then one is created (running the standard "sh") tied to a buffer named "shell'. In any case, the shell buffer becomes the current one and dot is positioned at the end of it. In that buffer output from the shell and programs run with it will appear. Anything typed into it will get sent to the subprocess when the *return* key is struck. This lets you interact with a shell using EMACS, and all of it's editing capability, as an intermediary. You can scroll backwards over a session, pick up pieces of text from other places and use them as input, edit while watching the execution of some program, and much more...
lisp	The *lisp* command is exactly the same as the *shell* command except that it starts up "cmulisp" in the "lisp" buffer. You can have both a shell and a lisp process going at the same time. You can even have as many shells going as you want, but this package doesn't support it.
grab-last-line	(ESC-=) This command takes the last string typed as input to the process and brings it back, as though you had typed it again. So if you muff a command, just type ESC-=, edit the line, and hit *return* again.
lisp-kill-output	(↑X↑K) [this only applies to *lisp* processes] Erases the output from the last command. If you don't want to see the output of the last command any more, just type ↑X↑K and it will go away.
pr-newline	(↑M -- return) Takes the text of the current line and sends it as input to the process tied to the current buffer. Actually, if dot is on the last line of the buffer, it takes the region from mark to the end of the buffer and sends it as input (output from a process causes the mark to be set after the inserted text); if dot is not on the last line, just the text of that line is shipped (presuming that your prompt is "$ ").
send-eot	(↑D) If dot is at the end of the buffer, then ↑D behaves just as it does outside of EMACS -- it sends an EOT to the subprocess (end of file to some folks). If dot isn't at the end of the buffer, then it does the usual character deletion.
send-int-signal	(\177 -- rubout) Sends an INT (Interrupt) signal to the subprocess, which should make it stop whatever it is doing.
send-quit-signal	(↑\) Sends a QUIT signal to the subprocess, making it stop whatever it is doing and produce a core dump.

20.15. pwd -- print and change the working directory

pwd	Prints the current working directory in the mode line, just like the shell command "pwd".
cd	Changes the current working directory, just like the shell command "cd". You should beware that *cd* only changes the current directory for EMACS, if it has already spawned a subprocess (a shell, for example) then a *cd* from within EMACS has no effect on the shell.

20.16. rmail -- a mail management system

EMACS may be used to send and receive electronic mail. The *rmail* command (Usually invoked as "ESC-Xrmail") is used for reading mail, *smail* is used for sending mail.

20.16.1. Sending Mail

When sending mail, either by using the *smail* command or from within *rmail*, EMACS constructs a buffer that contains an outline of the message to be sent and allows you to edit it. All that you have to do is fill in the blanks. When you exit from *smail* (by typing ↑C usually -- when you're editing the message body you will be in a recursive-edit) the message will be sent to the destinations and blindcopied to you. Several commands are available to help you in composing the message:

justify-paragraph	(ESC-j) Fixes up the line breaks in the current paragraph according to the current left and right margins.
exit-emacs	(↑C) Exits mail composition and attempts to send the mail. If all goes well the mail composition window will disappear and a confirmation message will appear at the bottom of the screen. If there is some sort of delivery error you will be placed back into the composition window and a message will appear. **Bug:** when delivery is attempted and there are errors in the delivery, the message will have been delivered to the acceptable addresses and not to the others. This makes retrying the message difficult since you have to manually eliminate the addresses to which the message has already been sent.
mail-abort-send	(↑X↑A) Aborts the message. If you're part-way through composing a message and decide that you don't want to send it, ↑X↑A will throw it away, after asking for confirmation.
mail-noblind-exit	(↑X↑C) Exits *smail* and send the message, just as ↑C will, except that a blind copy of the message will not be kept.
exit-emacs	(↑X↑F) Same as ↑C.
exit-emacs	(↑X↑S) Same as ↑C.
mail-append	(↑Xa) Positions dot at the end of the body and sets margins and abbrev tables appropriatly.
mail-cc	(↑Xc) Positions dot to the "cc:" field, creating it if necessary.
mail-insert	(↑Xi) Inserts the body of the message that was most recently looked at with rmail into the body of the message being composed. If, for instance, what you want to do is forward a message to someone, just read the message with rmail, then compose a message to the person you want to forward to, and type **↑Xi**.

mail-subject (↑Xs) Positions dot to the "subject:" field of the message.

mail-to (↑Xt) Positions dot to the "to:" field of the message.

20.16.2. Reading Mail

The *rmail* command provides a facility for reading mail from within EMACS. When it is running there are usually two windows on the screen: one shows a summary of all the messages in your mailbox and the other displays the "current" message. The summary window may contain something like this:

```
     02621525335022 29 Oct 1981  research!dmr    [empty]
   B 02621525335030 29 Oct 1981  =>Unix-Wizards  A plea for understanding
     02621525335040 31 Oct 1981  CSVAX.dmr       rc etymology
     02621525335072 3 Nov 1981   EHF             fyi
  A  02621352421000 3 Nov 1981   JIM             copyrights
   B 02621353040000 3 Nov 1981   =>JIM           Re: copyrights
     02621646433000 [empty]      [empty]         [empty]
   B 02621647417000 4 Nov 1981   =>research!ikey Emacs
>N   02622024522003 5 November   flaco           cooking class
```

This is broken into five columns, as indicated by the underlining.

- The first column contains some flags: '>' indicates the current message, 'B' indicates that the message is a blindcopy (ie. A copy of a message that you sent to someone else), 'A' indicates that you've answered the message, and 'N' indicates that the message is new.
- The second column contains a long string of digits that is internal information for the mail system.
- The third contains the date on which the mail was sent.
- The forth contains the sender of the message, unless it is a blindcopy, in which case it contains the destination (indicated by the "=>").
- The fifth column contains the subject of the message.

When in the summary window *Rmail* responds to the following commands:

rmail-shell (!) Puts you into a command shell so that you can execute Unix commands. Resume mail reading by typing ↑C.

execute-extended-command
(:) An emergency trap-door for executing arbitrary EMACS commands. You should never need this.

rmail-first-message
(<) Look at the first message in the message file.

rmail-last-message (>) Look at the last message in the message file.

rmail-help (?) Print a very brief help message

exit-emacs (↑C) Leave rmail. Changes marked in the message file directory (eg. deletions) will be made.

rmail-search-reverse	(**↑R**) Prompts for a search string and positions at the first message, scanning in reverse, whose directory entry contains the string.
rmail-search-forward	(**↑S**) Prompts for a search string and positions at the first message, scanning forward, whose directory entry contains the string.
rmail-append	(**a**) Append the current message to a file.
rmail-previous-page	(**b**) Moves backward in the window that contains the current message.
rmail-delete-message	(**d**) Flag the current message for deletion. It won't actually be deleted until you leave rmail.
rmail-next-page	(**f**) Moves forward in the window that contains the current message. To read a message that is longer than the window that contains it, just keep typing **f** and rmail will show you successive pages of it.
rmail-goto-message	(**g**) Moves to the *n*th message.
smail	(**m**) Lets you send some mail.
rmail-next-message	(**n**) Moves to the next message.
rmail-previous-message	(**p**) Moves to the previous message.
exit-emacs	(**q**) the same as **↑C**
rmail-reply	(**r**) Constructs a reply to the current message.
rmail-skip	(**s**) Moves to the *n*th message relative to this one.
rmail-undelete-message	(**u**) If the current message was marked for deletion, **u** removes that mark.

20.17. scribe -- weak assistance for dealing with Scribe documents

Scribe mode binds *justify-paragraph* to **ESC-j**, defines *appply-look* and binds it to **C-X-l**, turns on autofill, sets the right margin to 77 and updates the LastEditDate to the current date. It also binds *index-entry* to **ESC-I**, and *scribe-command* to **ESC-S**.

If the string "LastEditDate="" exists somewhere in the first 2000 characters of the document then then the region extending from it to the next "" is replaced by the current date and time. You're intended to stick in your document something like:

```
@String(LastEditDate="Thu Jul 15 17:10:56 1982")
```

EMACS will automatically maintain the date. The date will only change in the file you make some changes, the mere act of starting scribe-mode does not cause the date change to be permanent.

Apply-look reads a single character and then surrounds the current word with "@c[" and "]". So, if you've just typed "begin", typing **ESC-l-i** will change it to "@i[begin]", which appears in the document as "*begin*". This use of the word "look" comes from the *Bravo* text editor.

Index-entry takes a number of words and creates a Scribe index entry for that phrase, on a separate line. The current dot and mark are not modified. If the command is given with no prefix-argument, the current word is used as the index item. If a positive argument *n* is given, *n* words starting with the current word are used as the index phrase; a negative argument *n* causes the *n* words ending with the current word to be used. The easiest way to learn what the real rules are is to try it out; if you make a mistake, you can try again without having to change the cursor position, then delete the wrong index entries once you've got a right one.

Scribe-command is used to create a *Begin* -- *End* bracket pair for a specified scribe command. You are prompted for the name of the command (e.g., **Index, Itemize, Description**, etc.) For example, **ESC-S Itemize** would insert

```
@Begin(Itemize)

@End(Itemize)
```

and would leave the cursor on the blank line inside the begin--end brackets. If you always create scribe commands in this way, you'll never have unbalanced begin--ends in your scribe files.

20.18. scribe-bib -- Scribe bibliography creation mode

Scribe-bib mode provides a set of functions that create Scribe bibliography database entries. For each bibliography type scribe-bib mode provides a function that when executed prompts the user for appropriate fields and constructs a new entry of the proper type. The name of each of these functions is identical to the name of the corresponding bibliography type. Once the entry is created it can be edited using standard Emacs commands. The bibliography creation functions are invoked by name using ESC-X and are listed below:

@article	Create an @Article bibliography entry.
@book	Create an @Book bibliography entry.
@booklet	Create an @Booklet bibliography entry.
@inbook	Create an @InBook bibliography entry.
@incollection	Create an @InCollection bibliography entry.
@inproceedings	Create an @InProceedings bibliography entry.
@manual	Create an @Manual bibliography entry.
@mastersthesis	Create an @MastersThesis bibliography entry.

@misc	Create an @Misc bibliography entry.
@phdthesis	Create an @PhdThesis bibliography entry.
@proceedings	Create an @Proceedings bibliography entry.
@techreport	Create an @TechReport bibliography entry.
@unpublished	Create an @Unpublished bibliography entry.

20.19. spell -- a simple spelling corrector

The spell package implements the single function *spell.* It provides a simple facility for doing spelling correction. If you invoke *spell* it will scan your file looking for spelling errors, then it will go through a dialogue to let you fix them up. For each misspelled word EMACS will show you the word, some context around it and ask you what to do. If you type 'e' or '↑G' the spelling corrector will exit. If you type ' ' it will ignore the word. If you type 'r' it will ask for the text to use in replacing the word and perform a query-replace. ***Bug:*** This uses the Unix *spell* command which believes that its input is a source for the Unix standard text formatter troff/nroff; Spell misbehaves on Scribe .mss files.

20.20. srccom -- source comparison function

srccom	Compare text in two windows. To begin the comparison, place the dot at the beginning of one of the two pieces of text to be compared, switch to the other window, and place the dot at the beginning of the other piece of text. (If there are more than two windows, the two windows to be compared must be adjacent, and the dot must be left in the upper one.) When this command is invoked, it will search forward, stopping when either a difference is encountered or the end of the buffer is reached. *case-fold-search* governs comparison of case differences. The region is left around the equal portions in both windows.

20.21. tags -- a function tagger and finder

The tags package closely resembles the tags package found in Twenex EMACS. The database used by the tag package (called a tagfile) correlates function definitions to the file in which the definitions appear. The primary function of the tag package is to allow the user to specify the name of a function, and then have EMACS locate the definition of that function. The commands implemented are:

add-tag	Adds the current line (it should be the definition line for some function) to the current tagfile.
goto-tag	*goto-tag* takes a single string argument which is usually the name of a function and visits the file containing that function with the first line of the function at the top of the window. The string may actually be a substring of the function name (actually, any substring of the first line of the function definition). If *goto-tag* is given a numeric argument then rather than asking for a new string it will use the old string and search for the next occurrence of that string in the tagfile. This is used for stepping through a set of tags that contain the same string.

This is the most commonly used command in the tag package so it is often bound to a key: Twenex EMACS binds it to ESC-., but the Unix tag package doesn't bind it to anything, it presumes that the user will bind it (I use ↑X↑G).

make-tag-table	Takes a list of file names (with wildcards allowed) and builds a tagfile for all the functions in all of the files. It determines the language of the contents of the file from the extension. This command may take a while on large directories, be prepared to wait. A common use is to type "make-tag-table *.c".
recompute-all-tags	Goes through your current tag file and for each file mentioned refinds all of the tags. This is used to rebuild an entire tag file if you've made very extensive changes to the files mentioned and the tag package is no longer able to find functions. The tagfile contains *hints* to help the system locate the tagged function, as you make changes to the various files the hints become out of date. Periodically (no too often!) you should recompute the tagfile.
visit-function	Takes the function name at or before dot, does a *goto-tag* on that name, then puts you into a recursive-edit to look at the function definition. To get back to where you were, just type ↑C. This is used when you're editing something, have dot positioned at some function invocation, then want to look at the function.
visit-tag-table	Normally the name of the tagfile is ".tags" in the current directory. If you want to use some other tagfile, visit-tag-table lets you do that.

20.22. text-mode -- assist for simple text entry

Implements the *text-mode* command which ties ESC-j to *justify-paragraph* and sets up autofill with a left margin of 1 and a right margin of 77.

20.23. time -- a mode line clock

This package only implements one user-visible function, *time*, which puts the current time of day and load average (continuously updating!) in the mode line of each window. It uses global-mode-string and the subprocess control facility. Major!

20.24. transp -- transpose words or lines

The *transp* package allows transposition of word and lines (similar to the function of *transpose-character.*)

transpose-word	Takes the two words preceding *dot* and exchanges them. (If *dot* is within a word, it is counted as preceeding *dot.*)
transpose-line	Takes the two lines preceding *dot* and exchanges them. (If *dot* is within a line, it is counted as preceeding *dot.*)

There are also several global variables to control the *transpose-line* function:

&Default-Transpose-Direction

(default 1) Tells transpose-line which other line to transpose with the current on. If this is set to 1 (actually your favorite non-zero number will do) then transpose-line will use the line above the current one and if it is 0 transpose-line will use the line below the current one.

&Default-Transpose-Follow

(default 0) If this is set Non-zero it will cause transpose-line to leave the cursor(dot) on the line that got transposed, and if this is set to Zero it will stay at the same place in the file!

&Default-Transpose-Magic

(default 0) This variable controls some magic inside the transpose Line function. If it is set to zero, transpose-line will behave as controlled by the settings of the above variables. If this is set Non-Zero then the magic is controlled by the cursor position when transpose-line is invoked. If the cursor(dot) is somewhere in the middle of a line, then it behaves as if this variable were 0. If the cursor is at the end of a line, or at the beginning of a line, the magic will happen. If the cursor is at the beginning of the line transpose-line will override the above variable settings and assert that you want to transpose with the above line and that you want to follow the line you were on. If the cursor is at the end of a line transpose-line will assume that you want to transpose with the next line and that you want to follow the line you were on. The main reason for this magic is so that you can blip lines up and down in your buffer real easily.

20.25. undo -- undo previous commands

The **new-undo** command, which is usually bound to ↑X↑U allows the user to interactively undo the effects of previous commands. Typing ↑X↑U undoes the effects of the last command typed. It will then ask "Hit <space> to undo more", each <space> that you then hit will undo one more command. Typing anything but space will terminate undoing. If it is terminated with anything other than <return> the termination character will be executed just as though it were a normal command. **new-undo** is an undoable command, just like the others, so if you find that you've undone too much just type ↑X↑U again to undo the undo's.

20.26. writeregion -- write region to file

This package only implements one function, write-region-to-file, which takes the region between dot and mark and writes it to the named file.

21. Command Description

This chapter describes (in alphabetical order) all of the commands which are defined in the basic Unix EMACS system. Other commands may be defined by loading packages. Each description names the command and indicates the default binding.

! **[unbound]**

(! e_1) MLisp function that returns not e_1.

!= **[unbound]**

(!= e_1 e_2) MLisp function that returns true iff e_1 != e_2.

% **[unbound]**

(% e_1 e_2) MLisp function that returns e_1 % e_2 (the C mod operator).

& **[unbound]**

(& e_1 e_2) MLisp function that returns e_1 & e_2.

*** **[unbound]**

(* e_1 e_2) MLisp function that returns e_1 * e_2.

+ **[unbound]**

(+ e_1 e_2) MLisp function that returns e_1 + e_2.

- **[unbound]**

(- e_1 e_2) MLisp function that returns e_1 - e_2.

/ **[unbound]**

(/ e_1 e_2) MLisp function that returns e_1 / e_2.

< **[unbound]**

(< e_1 e_2) MLisp function that returns true iff e_1 < e_2.

<< **[unbound]**

(<< e_1 e_2) MLisp function that returns e_1 << e_2 (the C shift left operator).

<= **[unbound]**

(<= e_1 e_2) MLisp function that returns true iff e_1 <= e_2.

= **[unbound]**

(= e_1 e_2) MLisp function that returns true iff e_1 = e_2.

> **[unbound]**
(> e_1 e_2) MLisp function that returns true iff $e_1 > e_2$.

>= **[unbound]**
(>= e_1 e_2) MLisp function that returns true iff $e_1 >= e_2$.

>> **[unbound]**
(>> e_1 e_2) MLisp function that returns $e_1 >> e_2$ (the C shift right operator).

↑ **[unbound]**
(↑ e_1 e_2) MLisp function that returns $e_1 \uparrow e_2$ (the C XOR operator).

active-process **[unbound]**
(active-process) -- Returns the name of the active process as defined in the section describing the process mechanism.

append-region-to-buffer **[unbound]**
Appends the region between dot and mark to the named buffer. Neither the original text in the destination buffer nor the text in the region between dot and mark will be disturbed.

append-to-file **[unbound]**
Takes the contents of the current buffer and appends it to the named file. If the files doesn't exist, it will be created.

apropos **ESC-?**
Prompts for a keyword and then prints a list of those commands whose short description contains that keyword. For example, if you forget which commands deal with windows, just type "ESC-?`window`ESC".

arg **[unbound]**
(arg i [prompt]) evaluates to the i'th argument of the invoking function or prompts for it if called interactively [the prompt is optional, if it is omitted, the function cannot be called interactivly]. For example,

```
    (arg 1 "Enter a number: ")
```

Evaluates to the value of the first argument of the current function, if the current function was called from MLisp. If it was called interactively then it is prompted for. As another example, given:

```
    (defun (foo (+ (arg 1 "Number to increment? ") 1)))
```

then (foo 10) returns 11, but typing "ESC-Xfoo" causes emacs to ask "Number to increment? ". Language purists will no doubt cringe at this rather primitive parameter mechanism, but what-the-hell... it's amazingly powerful.

argc **[unbound]**

Is an MLisp function that returns the number of arguments that were passed to EMACS when it was invoked from the Unix shell. If either *argc* or *argv* are called early enough then EMACS's startup action of visiting the files named on the command line is suppressed.

argument-prefix ↑U

When followed by a string of digits ↑U causes that string of digits to be interpreted as a numeric argument which is generally a repetition count for the following command. For example, ↑U10↑N moves down 10 lines (the 10'th next). A string of *n* ↑U's followed by a command provides an argument to that command of 4^n. For example, ↑U↑N moves down four lines, and ↑U↑U↑N moves down 16. Argument-prefix should *never* be called from an MLisp function.

argv **[unbound]**

(argv *i*) returns the *i*th argument that was passed to EMACS when it was invoked from the Unix Shell. If EMACS were invoked as "emacs blatto" then (argv 1) would return the string "blatto". If either argc or argv are called early enough then EMACS's startup action of visiting the files named on the command line is suppressed.

auto-execute **[unbound]**

Prompt for and remember a command name and a file name pattern. When a file is read in via *visit-file* or *read-file* whose name matches the given pattern the given command will be executed. The command is generally one which sets the mode for the buffer. Patterns must be of the form "*string" or "string*": "*string" matches any filename whose suffix is "string"; "string*" matches any filename prefixed by "string". For example, **auto-execute c-mode ***.c will put EMACS into C mode for all files with the extension ".c".

autoload **[unbound]**

(autoload command file) defines the associated *command* to be autoloaded from the named *file*. When an attempt to execute the command is encountered, the file is loaded and then the execution is attempted again. the loading of the file must have redefined the command. Autoloading is useful when you have some command written in MLisp but you don't want to have the code loaded in unless it is actually needed. For example, if you have a function named box-it in a file named box-it.ml, then the command **(autoload "box-it" "box-it.ml")** will define the box-it command, but won't load its definition from box-it.ml. The loading will happen when you try to execute the box-it command.

backward-balanced-paren-line **[unbound]**

Moves dot backward until either

- The beginning of the buffer is reached.
- An unmatched open parenthesis, '(', is encountered. That is, unmatched between there and the starting position of dot.
- The beginning of a line is encountered at "parenthesis level zero". That is, without an unmatched ')' existing between there and the starting position of dot.

The definitions of parenthesis and strings from the syntax table for the current buffer are used.

backward-character **↑B**

Move dot backwards one character. Ends-of-lines and tabs each count as one character. You can't move back to before the beginning of the buffer.

backward-paragraph **ESC-[**

Moves to the beginning of the current or previous paragraph. Blank lines, and Scribe and nroff command lines separate paragraphs and are not parts of paragraphs.

backward-paren **[unbound]**

Moves dot backward until an unmatched open parenthesis, '(', or the beginning of the buffer is found. This can be used to aid in skipping over Lisp S-expressions. The definitions of parenthesis and strings from the syntax table for the current buffer are used.

backward-sentence **ESC-A**

Move dot backward to the beginning of the preceeding sentence; if dot is in the middle of a sentence, move to the beginning of the current sentence. Sentences are seperated by a '.', '?' or '!' followed by whitespace.

backward-word **ESC-B**

If in the middle of a word, go to the beginning of that word, otherwise go to the beginning of the preceding word. A word is a sequence of alphanumerics.

baud-rate **[unbound]**

An MLisp function that returns what EMACS thinks is the baud rate of the communication line to the terminal. The baud rate is (usually) 10 times the number of characters transmitted ber second. (Baud-rate) can be used for such things as conditionally setting the display-file-percentage variable in your EMACS profile: (setq display-file-percentage (> (baud-rate) 600))

beginning-of-file **ESC-<**

Move dot to just before the first character of the current buffer.

beginning-of-line **↑A**

Move dot to the beginning of the line in the current buffer that contains dot; that is, to just after the preceeding end-of-line or the beginning of the buffer.

beginning-of-window **ESC-,**

Move dot to just in front of the first character of the first line displayed in the current window.

bind-to-key **[unbound]**

Bind a named macro or procedure to a given key. All future hits on the key will cause the named macro or procedure to be called. The key may be a control key, and it may be prefixed by ↑X or ESC. For example, if you want ESC-= to behave the way ESC-X*print* does, then typing ESC-X*bind-to-key print* ESC-= will do it.

bobp **[unbound]**

(bobp) is an MLisp predicate which is true iff dot is at the beginning of the buffer.

bolp **[unbound]**

(bolp) is an MLisp predicate which is true iff dot is at the beginning of a line.

buffer-size **[unbound]**

(buffer-size) is an MLisp function that returns the number of characters in the current buffer.

c-mode **[unbound]**

Incompletely implemented.

c= **[unbound]**

(c= e_1 e_2) MLisp function that returns true iff e_1 is equal to e_2 taking into account the character translations indicated by case-fold-search. If word-mode-search is in effect, then upper case letters are "c=" to their lower case equivalents.

case-region-capitalize **[unbound]**

Capitalize all the words in the region between dot and mark by making their first characters upper case and all the rest lower case.

case-region-invert **[unbound]**

Invert the case of all alphabetic characters in the region between dot and mark.

case-region-lower **[unbound]**

Change all alphabetic characters in the region between dot and mark to lower case.

case-region-upper **[unbound]**

Change all alphabetic characters in the region between dot and mark to upper case.

case-word-capitalize **[unbound]**
Capitalize the current word (the one above or to the left of dot) by making its first character upper case and all the rest lower case.

case-word-invert **[unbound]**
Invert the case of all alphabetic characters in the current word (the one above or to the left of dot).

case-word-lower **[unbound]**
Change all alphabetic characters in the current word (the one above or to the left of dot) to lower case.

case-word-upper **[unbound]**
Change all alphabetic characters in the current word (the one above or to the left of dot) to upper case.

change-current-process **[unbound]**
(change-current-process "process-name") -- Sets the current process to the one named.

change-directory **[unbound]**
Changes the current directory (for EMACS) to the named directory. All future file write and reads (↑X↑S, ↑X↑V, etc.) will be interpreted relative to that directory.

char-to-string **[unbound]**
Takes a numeric argument and returns a one character string that results from considering the number as an ascii character.

checkpoint **[unbound]**
Causes all modified buffers with an out of date checkpoint file to be checkpointed. This function is normally called automatically every *checkpoint-frequency* keystrokes.

Command prefix, also known as META **ESC**
The next character typed will be interpreted as a command based on the fact that it was preceded by ESC. The name meta for the ESC character comes from funny keyboards at Stanford and MIT that have a Meta-shift key which is used to extend the ASCII character set. Lacking a Meta key, we make do with prefixing with an ESC character. You may see (and hear) commands like ESC-V referred to as Meta-V. Sometimes the ESC key is confusingly written as $, so ESC-V would be written as $V. ESC is also occasionally referred to as *Altmode*, from the labeling of a key on those old favorites, model 33 teletypes.

command-prefix **↑X**

The next character typed will be interpreted as a command based on the fact that it was preceded by **↑X.**

compile-it **↑X↑E**

Make is a standard Unix program which takes a description of how to compile a set of programs and compiles them. The output of *make* (and the compilers it calls) is placed in a buffer which is displayed in a window. If any errors were encountered, EMACS makes a note of them for later use with **↑X↑N**. Presumably, a data base has been set up for *make* that causes the files which have been edited to be compiled. **↑X↑E** then updates the files that have been changed and *make* does the necessary recompilations, and EMACS notes any errors and lets you peruse them with **↑X↑N.**

If **↑X↑E** is given a non-zero argument, then rather than just executing *make* EMACS will prompt for a Unix command line to be executed. Modified buffers will still be written out, and the output will still go to the *Error log* buffer and be parsed as error messages for use with **↑X↑N**. One of the most useful applications of this feature involves the *grep* program. "**↑U↑X↑E**grep -n MyProc *.cESC" will scan through all C source files looking for the string "MyProc" (which could be the name of a procedure). You can then use **↑X↑N** to step through all places in all the files where the string was found. **Note:** The version of *grep* in my bin directory, /usr/jag/bin/grep, must be used: it prints line numbers in a format that is understood by EMACS. (ie. "*FileName*, line *LineNumber*)

concat **[unbound]**

Takes a set of string arguments and returns their concatenation.

continue-process **[unbound]**

(continue-process "process-name") -- Continue a process stopped by *stop-process.*

copy-region-to-buffer **[unbound]**

Copies the region between dot and mark to the named buffer. The buffer is emptied before the text is copied into it; the region between dot and mark is left undisturbed.

current-buffer-name **[unbound]**

MLisp function that returns the current buffer name as a string.

current-column **[unbound]**

(current-column) is an MLisp function that returns the printing column number of the character immediately following dot.

current-file-name **[unbound]**
MLisp function that returns the file name associated with the current buffer as a string. If there is no associated file name, the null string is returned.

current-indent **[unbound]**
(current-indent) is an MLisp function the returns the amount of whitespace at the beginning of the line which dot is in (the printing column number of the first non-whitespace character).

current-process **[unbound]**
(current-process) -- Returns the name of the current process as defined in the section describing the process mechanism.

current-time **[unbound]**
MLisp function that returns the current time of day as a string in the format described in CTIME(3), with the exception that the trailing newline will have been stripped off. (substr (current-time) -4 4) is the current year.

declare-buffer-specific **[unbound]**
Takes a list of variables and declares them to have *buffer-specific* values. A buffer-specific variable has a distinct instance for each buffer in existance and a default value which is used when new buffers are created. When a buffer-specific variable is assigned a value only the instance associated with the currently selected buffer is affected. To set the default value for a buffer-specific variable, use *setq-default* or *set-default.* Note that if you have a global variable which is eventually declared buffer-specific then the global value becomes the default.

declare-global **[unbound]**
Takes a list of variables and for each that is not already bound a global binding is created. Global bindings outlive all function calls.

define-buffer-macro **[unbound]**
Take the contents of the current buffer and define it as a macro whose name is associated with the buffer. This is how one redefines a macro that has been edited using edit-macro.

define-global-abbrev **[unbound]**
Define (or redefine) an abbrev with the given name for the given phrase in the global abbreviation table.

define-hooked-global-abbrev **[unbound]**
The commands *define-hooked-global-abbrev* and *define-hooked-local-abbrev* behave exactly as the unhooked versions do (*define-global-abbrev* and *define-local-abbrev*) except that they also associate a named command with the abbrev. When the abbrev triggers, rather than replacing the abbreviation with the expansion phrase the hook procedure is invoked. The character that trigged the abbrev will not have been inserted, but will be inserted immediatly after the hook procedure returns [unless the procedure returns 0]. The abbreviation will be the word immediatly to the left of dot, and the function *abbrev-expansion* returns the phrase that the abbrev would have expanded to.

define-hooked-local-abbrev **[unbound]**
See the description of *define-hooked-global-abbrev*.

define-keyboard-macro **[unbound]**
Give a name to the current keyboard macro. A keyboard macro is defined by using the ↑X(and ↑X) command; define-keyboard-macro takes the current keyboard macro, squirrels it away in a safe place, gives it a name, and erases the keyboard macro. define-string-macro is another way to define a macro.

define-keymap **[unbound]**
(define-keymap "mapname") defines a new, empty, keymap with the given name. See the section on keymaps, 15 page 17, for more information.

define-local-abbrev **[unbound]**
Define (or redefine) an abbrev with the given name for the given phrase in the local abbreviation table. A local abbrev table must have already been set up with *use-abbrev-table*.

define-string-macro **[unbound]**
Define a macro given a name and a body as a string entered in the minibuffer. Note: to get a control character into the body of the macro it must be quoted with ↑Q. define-keyboard-macro is another way to define a macro.

defun **[unbound]**
(defun (name expressions...)...) is an MLisp function that defines a new MLisp function with the given name and a body composed of the given expressions. The value of the function is the value of the last expression. For example:

```
(defun
    (indent-line            ; this function just sticks a tab at
        (save-excursion     ; the beginning of the current line
            (beginning-of-line) ; without moving dot.
            (insert-string "    ")
        )
    )
)
```

delete-buffer **[unbound]**
Deletes the named buffer.

delete-macro **[unbound]**
Delete the named macro.

delete-next-character **↑D**
Delete the character immediatly following dot; that is, the character on which the terminals cursor sits. Lines may be merged by deleting newlines.

delete-next-word **ESC-D**
Delete characters forward from dot until the next end of a word. If dot is currently not in a word, all punctuation up to the beginning of the word is deleted as well as the word.

delete-other-windows **↑X1**
Go back to one-window mode. Generally useful when EMACS has spontaneously generated a window (as for ESC-? or ↑X↑B) and you want to get rid of it.

delete-previous-character **↑H**
Delete the character immediatly preceding dot; that is, the character to the left of the terminals cursor. If you've just typed a character, ↑H (*backspace*) will delete it. Lines may be merged by deleting newlines.

delete-previous-character **RUBOUT**
Delete the character immediatly preceding dot; that is, the character to the left of the terminals cursor. If you've just typed a character, RUBOUT will delete it. Lines may be merged by deleting newlines.

delete-previous-word **ESC-H**
If not in the middle of a word, delete characters backwards (to the left) until a word is found. Then delete the word to the left of dot. A word is a sequence of alphanumerics.

delete-region-to-buffer **ESC-↑W**
Wipe (kill, delete) all characters between dot and the mark. The deleted text is moved to a buffer whose name is prompted for, which is emptied first.

delete-to-killbuffer **↑W**
Wipe (kill, delete) all characters between dot and the mark. The deleted text is moved to the kill buffer, which is emptied first.

delete-white-space **[unbound]**

Deletes all whitespace characters (spaces and tabs) on either side of dot.

delete-window **↑XD**

Removes the current window from the screen and gives it's space to it's neighbour below (or above) and makes the current window and buffer those of the neighbour.

describe-bindings **[unbound]**

Places in the *Help* window a list of all the keys and the name of the procedure that they are bound to. This listing is suitable for printing and making you own quick-reference card for your own customized version of EMACS.

describe-command **[unbound]**

Uses the Info system to describe some named command. You will be prompted in the minibuf for the name of a command and then Info will be invoked to show you the manual entry describing it. You can then use Info to browse around, or simply type ↑C to resume editing.

describe-key **[unbound]**

Describe the given key. ESC-X**describe-key** ESC-X will print a short descrition of the ESC-X key. It tells you the name of the command to which the key is bound. To find out more about the command, use *describe-command.*

describe-variable **[unbound]**

Uses the Info system to describe some named variable. You will be prompted in the minibuf for the name of a variable and then Info will be invoked to show you the manual entry describing it. You can then use Info to browse around, or simply type ↑C to resume editing.

describe-word-in-buffer **↑X↑D**

Takes the word nearest the cursor and looks it up in a data base and prints the information found. This data base contains short one-line descriptions of all of the Unix standard procedures and Franz Lisp standard functions. The idea is that if you've just typed in the name of some procedure and can't quite remember which arguments go where, just type ↑X↑D and EMACS will try to tell you.

digit **[unbound]**

Heavy wizardry: you don't want to know. "digit" should eventually disappear.

dot **[unbound]**
(dot) is an MLisp function that returns the number of characters to the left of dot plus 1 (ie. if dot is at the beginning of the buffer, (dot) returns 1). The value of the function is an object of type "marker" -- if it is assigned to a variable then as changes are made to the buffer the variable's value continues to indicate the same position in the buffer.

dump-syntax-table **[unbound]**
Dumps a readable listing of a syntax table into a buffer and makes that buffer visible.

edit-macro **[unbound]**
Take the body of the named macro and place it in a buffer called *Macro edit.* The name of the macro is associated with the buffer and appears in the information bar at the bottom of the window. The buffer may be edited just like any other buffer (this is, in fact, the intent). After the macro body has been edited it may be redefined using *define-buffer-macro.*

emacs-version **[unbound]**
Returns a string that describes the current EMACS version.

end-of-file **ESC->**
Move dot to just after the last character of the buffer.

end-of-line **↑E**
Move dot to the end of the line in the current buffer that contains dot; that is, to just after the following end-of-line or the end of the buffer.

end-of-window **ESC-.**
Move dot to just after the last character visible in the window.

enlarge-window **↑XZ**
Makes the current window one line taller, and the window below (or the one above if there is no window below) one line shorter. Can't be used if there is only one window on the screen.

eobp **[unbound]**
(eobp) is an MLisp predicate that is true iff dot is at the end of the buffer.

eolp **[unbound]**
(eolp) is an MLisp predicate that is true iff dot is at the end of a line.

eot-process **[unbound]**
(eot-process "process-name") -- Send an EOT to the process.

erase-buffer **[unbound]**
Deletes all text from the current buffer. Doesn't ask to make sure if you really want to do it.

erase-region **[unbound]**
Erases the region between dot and mark. It is like *delete-to-killbuffer* except that it doesn't move the text to the kill buffer.

error-message **[unbound]**
(error-message "string-expressions") Sends the *string-expressions* to the screen as an error message where it will appear at the bottom of the screen. EMACS will return to keyboard level, unless caught by *error-occured.*

error-occured **[unbound]**
(error-occured expressions...) executes the given expressions and ignores their values. If all executed successfully, error-occured returns false. Otherwise it returns true and all expressions after the one which encountered the error will not be executed.

exchange-dot-and-mark **↑X↑X**
Sets dot to the currently marked position and marks the old position of dot. Useful for bouncing back and forth between two points in a file; particularly useful when the two points delimit a region of text that is going to be operated on by some command like ↑W (erase region).

execute-extended-command **ESC-X**
EMACS will prompt in the minibuffer (the line at the bottom of the screen) for a command from the extended set. These deal with rarely used features. Commands are parsed using a Twenex style command interpreter: you can type ESC or **space** to invoke command completion, or '?' for help with what you're allowed to type at that point. This doesn't work if it's asking for a key or macro name.

execute-keyboard-macro **↑XE**
Takes the keystrokes remembered with **↑X(** and **↑X)** and treats them as though they had been typed again. This is a cheap and easy macro facility. For more power, see the define-string-macro, define-keyboard-macro and bind-to-key commands.

execute-mlisp-buffer **[unbound]**
Parse the current buffer as as a single MLisp expression and execute it. This is what is generally used for testing out new functions: stick your functions in a buffer wrapped in a *defun* and use execute-mlisp-buffer to define them.

execute-mlisp-line **ESC-ESC**

Prompt for a string, parse it as an MLisp expression and execute it.

execute-monitor-command **↑X!**

Prompt for a Unix command then execute it, placing its output into a buffer called *Command execution* and making that buffer visible in a window. The command will not be able to read from its standard input (it will be connected to /dev/null). For now, there is no way to execute an interactive subprocess.

exit-emacs **↑C**

Exit EMACS. Will ask if you're sure if there are any buffers that have been modified but not written out.

exit-emacs **↑X↑C**

Exit EMACS. Will ask if you're sure if there are any buffers that have been modified but not written out.

exit-emacs **ESC-↑C**

Exit EMACS. Will ask if you're sure if there are any buffers that have been modified but not written out.

expand-file-name **[unbound]**

Takes a string representing a file name and expands it into an absolute pathname. For example, if the current directory is "/usr/frodo" then **(expand-file-name "../bilbo")** will return "/usr/bilbo".

expand-mlisp-variable **[unbound]**

Prompts for the name of a declared variable then inserts the name as text into the current buffer. This is very handly for typing in MLisp functions. It's also fairly useful to bind it to a key for easy access.

expand-mlisp-word **[unbound]**

Prompt for the name of a command then insert the name as text into the current buffer. This is very handly for typing in MLisp functions. It's also fairly useful to bind it to a key for easy access.

extend-database-search-list **[unbound]**

(extend-database-search-list dbname filename) adds the given data base file to the data base search list (dbname). If the database is already in the search list then it is left, otherwise the new database is added at the beginning of the list of databases.

fetch-database-entry **[unbound]**

(fetch-database-entry dbname key) takes the entry in the data base corresponding to the given key and inserts it into the current buffer.

file-exists **[unbound]**

(file-exists *fn*) returns 1 if the file named by *fn* exists and is writable, 0 if it does not exist, and -1 if it exists and is readable but not writable.

filter-region **[unbound]**

Take the region between dot and mark and pass it as the standard input to the given command line. Its standard output replaces the region between dot and mark. Use this to run a region through a Unix style-filter.

following-char **[unbound]**

(following-char) is an MLisp function that returns the character immediatly following dot. The null character (0) is returned if dot is at the end of the buffer. Remember that dot is not 'at' some character, it is between two characters.

forward-balanced-paren-line **[unbound]**

Moves dot forward until either

- The end of the buffer is reached.
- An unmatched close parenthesis, ')', is encountered. That is, unmatched between there and the starting position of dot.
- The beginning of a line is encountered at "parenthesis level zero". That is, without an unmatched '(' existing between there and the starting position of dot.

The definitions of parenthesis and strings from the syntax table for the current buffer are used.

forward-character **↑F**

Move dot forwards one character. Ends-of-lines and tabs each count as one character. You can't move forward to after the end of the buffer.

forward-paragraph **ESC-]**

Moves to the end of the current or following paragraph. Blank lines, and Scribe and nroff command lines separate paragraphs and are not parts of paragraphs.

forward-paren **[unbound]**

Moves dot forward until an unmatched close parenthesis, ')', or the end of the buffer is found. This can be used to aid in skipping over Lisp S-expressions. The definitions of parenthesis and strings from the syntax table for the current buffer are used.

forward-sentence **ESC-E**

Move dot forward to the beginning of the next sentence. Sentences are seperated by a '.', '?' or '!' followed by whitespace.

forward-word **ESC-F**

Move dot forward to the end of a word. If not currently in the middle of a word, skip all intervening punctuation. Then skip over the word, leaving dot positioned after the last character of the word. A word is a sequence of alphanumerics.

get-tty-buffer **[unbound]**

Given a prompt string it reads the name of a buffer from the tty using the minibuf and providing command completion.

get-tty-character **[unbound]**

Reads a single character from the terminal and returns it as an integer. The cursor is not moved to the message area, it is left in the text window. This is useful when writing things like query-replace and incremental search.

get-tty-command **[unbound]**

(get-tty-command *prompt*) prompts for the name of a declared function (using command completion & providing help) and returns the name of the function as a string. For example, the *expand-mlisp-word* function is simply (insert-string (get-tty-command ": expand-mlisp-word ")).

get-tty-string **[unbound]**

Reads a string from the terminal using its single string parameter for a prompt. Generally used inside MLisp programs to ask questions.

get-tty-variable **[unbound]**

(get-tty-variable *prompt*) prompts for the name of a declared variable (using command completion & providing help) and returns the name of the variable as a string. For example, the *expand-mlisp-variable* function is simply (insert-string (get-tty-variable ": expand-mlisp-variable ")).

getenv **[unbound]**

(getenv "*varname*") returns the named shell environment variable. for example, (getenv "HOME") will return a string which names your home directory.

global-binding-of **[unbound]**

Returns the name of the procedure to which a keystroke sequence is bound in the global keymap. "nothing" is returned if the sequence is unbound. The procedure *local-binding-of* performs a similar function for the local keymap.

goto-character **[unbound]**

Goes to the given character-position. (goto-character 5) goes to character position 5.

if **[unbound]**

(if test thenclause elseclause) is an MLisp function that executes and returns the value of *thenclause* iff *test* is true; otherwise it executes *elseclause* if it is present. For example:

```
(if (eolp)
    (to-col 33)
)
```

will tab over to column 33 if dot is currently at the end of a line.

illegal-operation **[unbound]**

Illegal-operation is bound to those keys that do not have a defined interpretation. Executing illegal-operation is an error. Most notably, ↑G, ESC-↑G, ↑X↑G are bound to *illegal-opetation* by default, so that typing ↑G will always get you out of whatever strange state you are in.

indent-C-procedure **ESC-J**

Take the current C procedure and reformat it using the *indent* program, a fairly sophisticated pretty printer. *Indent-C-procedure* is God's gift to those who don't like to fiddle about getting their formatting right. *Indent-C-procedure* is usually bound to ESC-J. When switching from mode to mode, ESC-J will be bound to procedures appropriate to that mode. For example, in text mode ESC-J is bound to *justify-paragraph.*

insert-character **[unbound]**

Inserts its numeric argument into the buffer as a single character. (insert-character '0') inserts the character '0' into the buffer.

insert-file **↑X↑I**

Prompt for the name of a file and insert its contents at dot in the current buffer.

insert-filter **[unbound]**

Insert a filter-procedure between a process and EMACS. This function should subsume the *start-filtered-process* function, but we should retain that one for compatibility I suppose...

insert-string **[unbound]**
(insert-string stringexpressions) is an MLisp function that inserts the strings that result from evaluating the given *stringexpressions* and inserts them into the current buffer just before dot.

int-process **[unbound]**
(int-process "process-name") -- Send an interrupt signal to the process.

interactive **[unbound]**
An MLisp function which is true iff the invoking MLisp function was invoked interactively (ie. bound to a key or by ESC-X).

is-bound **[unbound]**
an MLisp predicate that is true iff all of its variable name arguments are bound.

justify-paragraph **[unbound]**
Take the current paragraph (bounded by blank lines or Scribe control lines) and pipe it through the "fmt" command which does paragraph justification. *justify-paragraph* is usually bound to ESC-J when in text mode.

kill-process **[unbound]**
(kill-process "process-name") -- Send a kill signal to the process.

kill-to-end-of-line ↑K
Deletes characters forward from dot to the immediatly following end-of-line (or end of buffer if there isn't an end of line). If dot is positioned at the end of a line then the end-of-line character is deleted. Text deleted by the ↑K command is placed into the *Kill buffer* (which really is a buffer that you can look at). A ↑K command normally erases the contents of the kill buffer first; subsequent ↑K's in an unbroken sequence append to the kill buffer.

last-key-struck **[unbound]**
The last command character struck. If you have a function bound to many keys the function may use last-key-struck to tell which key was used to invoke it. (insert-character (last-key-struck)) does the obvious thing.

length **[unbound]**
Returns the length of its string parameter. (length "time") => 4.

line-to-top-of-window **ESC-!**

What more can I say? This one is handy if you've just searched for the declaration of a procedure, and want to see the whole body (or as much of it as possible).

list-buffers **↑X↑B**

Produces a listing of all existing buffers giving their names, the name of the associated file (if there is one), the number of characters in the buffer and an indication of whether or not the buffer has been modified since it was read or written from the associated file.

list-databases **[unbound]**

(list-databases) lists all data base search lists.

list-processes **[unbound]**

(list-processes) -- Analagous to "list-buffers". Processes which have died only appear once in this list before completely disappearing.

load **[unbound]**

Read the named file as a series of MLisp expressions and execute them. Typically a loaded file consists primarily of *defun*'s and buffer-specific variable assignments and key bindings. *Load* is usually used to load macro libraries and is used to load ".emacs_pro" from your home directory when EMACS starts up.

For example, loading this file:

```
(setq right-margin 75)
(defun (my-linefeed
            (end-of-line)
            (newline-and-indent)
       )
)
(bind-to-key "my-linefeed" 10)
```

sets the *right-margin* to 75 and defines a function called *my-linefeed* and binds it to the linefeed key (which is the ascii character 10 (decimal))

The file name given to *load* is interpreted relative to the EPATH environment variable, which is interpreted in the same manner as the shell's PATH variable. That is, it provides a list of colon-separated names that are taken to be the names of directories that are searched for the named files. The default value of EPATH searches your current directory and then a central system directory.

Temporary hack: in previous versions of EMACS *load*ed files were treated as a sequence of keystrokes. This behaviour has been decreed bogus and unreasonable, hence it has been changed. However, to avoid loud cries of anguish the *load* command still exhibits the old behaviour if the first character of the loaded file is an **ESC.**

local-bind-to-key **[unbound]**

Prompt for the name of a command and a key and bind that command to the given key but unlike *bind-to-key* the binding only has effect in the current buffer. This is generally used for mode specific bindings that will generally differ from buffer to buffer.

local-binding-of **[unbound]**

Returns the name of the procedure to which a keystroke sequence is bound in the local keymap. "nothing" is returned if the sequence is unbound. The procedure *global-binding-of* performs a similar function for the global keymap.

looking-at **[unbound]**

(looking-at "SearchString") is true iff the given regular expression search string matches the text immediatly following dot. This is for use in packages that want to do a limited sort of parsing. For example, if dot is at the beginning of a line then **(looking-at "[\t]*else])** will be true if the line starts with an "else". See section 14, page 15 for more information on regular expressions.

mark **[unbound]**

An MLisp function that returns the position of the marker in the current buffer. An error is signaled if the marker isn't set. The value of the function is an object of type "marker" -- if it is assigned to a variable then as changes are made to the buffer the variable's value continues to indicate the same position in the buffer.

message **[unbound]**

(message stringexpressions) is an MLisp function that places the strings that result from the evaluation of the given *stringexpressions* into the message region on the display (the line at the bottom).

modify-syntax-entry **[unbound]**

Modify-syntax-entry is used to modify a set of entries in the syntax table associated with the current buffer. Syntax tables are associated with buffers by using the *use-syntax-table* command. Syntax tables are used by commands like *forward-paren* to do a limited form of parsing for language dependent routines. They define such things as which characters are parts of words, which quote strings and which delimit comments (currently, nothing uses the comment specification). To see the contents of a syntax table, use the *dump-syntax-table* command.

The parameter to *modify-syntax-entry* is a string whose first five characters specify the interpretation of the sixth and following characters.

The first character specifies the type. It may be one of the following:

'w'	A word character, as used by such commands as *forward-word* and *case-word-capitalize.*
space	A character with no special interpretation.
'('	A left parenthesis. Typical candidates for this type are the characters '(', '[' and '{'. Characters of this type also have a matching right parenthesis specified (')', ']' and '}' for example) which appears as the second character of the parameter to *modify-syntax-entry.*

')'	A right parenthesis. Typical candidates for this type are the characters ')', ']' and '}'. Characters of this type also have a matching left parenthesis specified ('(', '[' and '{' for example) which appears as the second character of the parameter to *modify-syntax-entry.*
'"'	A quote character. The C string delimiters " and ' are usually given this class, as is the Lisp |.
'\'	A prefix character, like \ in C or / in MacLisp.

The second character of the parameter is the matching parenthesis if the character is of the left or right parenthesis type. If you specify that '(' is a right parenthesis matched by ')', then you should also specify that ')' is a left parenthesis matched by '('.

The third character, if equal to '{', says that the character described by this syntax entry can begin a comment; the forth character, if equal to '}' says that the character described by this syntax entry can end a comment. If either the beginning or ending comment sequence is two characters long, then the fifth character provides the second character of the comment sequence.

The sixth and following characters specify which characters are described by this entry; a range of characters can be specified by putting a '-' between them, a '-' can be described if it appears as the sixth character.

A few examples, to help clear up my muddy exposition:

```
(modify-syntax-entry "w    -") ; makes '-' behave as a normal word
                               ; character (ESC-F will consider
                               ; one as part of a word)
(modify-syntax-entry "()   [") ; makes '[' behave as a left parenthesis
                               ; which is matched by ']'
(modify-syntax-entry ")[   ]") ; makes ']' behave as a right parenthesis
                               ; which is matched by '['
```

move-dot-to-x-y **[unbound]**

(move-dot-to-x-y *x y*) switches to the buffer and sets dot to the positon of the character that was displayed at screen coordinates *x,y*. If *x* and *y* don't point to a valid character (eg. if they are out of bounds or point to a mode line) an error is flagged.

This function is intended for use supporting mice and tablets. One way to do this is to have depressions of the tablet button generate a sequence of keystrokes that EMACS sees as normal tty input. If, for example, the tablet was to transmit the four charcters ESC-M-*x*-*y* when the button was depressed over character *x,y* then the following function would provide simple support for it:

```
(defun (mouse-set-dot x y
            (setq x (get-tty-character))
            (setq y (get-tty-character))
            (move-dot-to-x-y x y)
       ))

(bind-to-key "mouse-set-dot" "\eM")
```

move-to-comment-column **[unbound]**

If the cursor is *not* at the beginning of a line, **ESC-C** moves the cursor to the column specified by the comment-column variable by inserting tabs and spaces as needed. In any case, it the sets the right margin to the column finally reached. This is usually used in macros for language-specific comments.

nargs **[unbound]**

An MLisp function which returns the number of arguments passed to the invoking MLisp function. For example, within the execution of *foo* invoked by (foo x y) the value of *nargs* will be 2.

narrow-region **[unbound]**

The *narrow-region* command sets the restriction to encompass the region between dot and mark. Text outside this region will henceforth be totally invisible. It won't appear on the screen and it won't be manipulable by any editing commands. This can be useful, for instance, when you want to perform a replacement within a few paragraphs: just narrow down to a region enclosing the paragraphs and execute *replace-string.*

newline **[unbound]**

Just inserts a newline character into the buffer -- this is what the RETURN (↑M) key is generally bound to.

newline-and-backup **↑O**

Insert an end-of-line immediatly *after* dot, effectivly opening up space. If dot is positioned at the beginning of a line, then **↑O** will create a blank line preceding the current line and position dot on that new line.

newline-and-indent LINEFEED

Insert a newline, just as typing **RETURN** does, but then insert enough tabs and spaces so that the newly created line has the same indentation as the old one had. This is quite useful when you're typing in a block of program text, all at the same indentation level.

next-error **↑X↑N**

Take the next error message (as returned from the **↑X↑E** (compile) command), do a *visit* (**↑X↑V**) on the file in which the error occurred and set dot to the line on which the error occurred. The error message will be displayed at the top of the window associated with the *Error log* buffer.

next-line **↑N**

Move dot to the next line. **↑N** and **↑P** attempt to keep dot at the same horizontal position as you move from line to line.

next-page **↑V**

Reposition the current window on the current buffer so that the next page of the buffer is visible in the window (where a *page* is a group of lines slightly smaller than a window). In other words, it flips you forward a page in the buffer. Its inverse is ESC-V. If possible, dot is kept where it is, otherwise it is moved to the middle of the new page.

next-window **↑XN**

Switches to the window (and associated buffer) that is below the current window.

nothing **[unbound]**

Nothing evaluates the same as novalue (ie. it returns a void result) except that if it is bound to some key or attached to some hook then the key or hook behave as though no command was bound to them. For example, if you want to remove the binding of a single key, just bind it to "nothing".

novalue **[unbound]**

Does nothing. (novalue) is a complete no-op, it performs no action and returns no value. Generally the value of a function is the value of the last expression evaluated in it's body, but this value may not be desired, so (novalue) is provided so that you can throw it away.

page-next-window **ESC-↑V**

Repositions the window below the current one (or the top one if the current window is the lowest one on the screen) on the displayed buffer so that the next page of the buffer is visible in the window (where a *page* is a group of lines slightly smaller than a window). In other words, it flips you forward a page in the buffer of the *other* window.

If ESC-↑V is given an argument it will flip the buffer backwards a page, rather than forwards. So ESC-↑V is roughly equivalent to ↑V and ↑UESC-↑V is roughly equivalent to ESC-V except that they deal with the other window. Yes, yes, yes. I realize that this is a bogus command structure, but I didn't invent it. Besides, you can learn to love it.

parse-error-messages-in-region **[unbound]**

Parses the region between dot and mark for error messages (as in the *compile-it* (↑X↑E) command) and sets up for subsequent invocations of *next-error* (↑X↑N). See the description of the *compile-it* command, and section 10 (page 6).

pause-emacs **[unbound]**

Pause, giving control back to the superior shell using the job control facility of Berkeley Unix. The screen is cleaned up before the shell regains control, and when the shell gives control back to EMACS the screen will be fixed up again. Users of the sea-shell (csh) will probably rather use this command than "return-to-monitor", which is similar, except that it recursivly invokes a new shell.

pop-to-buffer **[unbound]**
Switches to a buffer whose name is provided and ties that buffer to a popped-up window. Pop-to-buffer is exactly the same as switch-to-buffer except that switch-to-buffer ties the buffer to the current window, pop-to-buffer finds a new window to tie it to.

preceding-char **[unbound]**
(preceding-char) is an MLisp function that returns the character immediatly preceding dot. The null character (0) is returned if dot is at the beginning of the buffer. Remember that dot is not 'at' some character, it is between two characters.

prefix-argument-loop **[unbound]**
(prefix-argument-loop <statements>) executes <statements> prefix-argument times. Every function invocation is always prefixed by some argument, usually by the user typing ↑U*n*. If no prefix argument has been provided, 1 is assumed. See also the command provide-prefix-argument and the variable prefix-argument.

prepend-region-to-buffer **[unbound]**
Prepends the region between dot and mark to the named buffer. Neither the original text in the destination buffer nor the text in the region between dot and mark will be disturbed.

previous-command **[unbound]**
`(previous-command)` *usually* returns the character value of the keystroke that invoked the previous command. In is something like *last-key-struck*, which returns the keystroke that invoked the current command. However, a function may set the variable *this-command* to some value, which will be the value of *previous-command* after the next command invocation. This rather bizarre command/variable pair is intended to be used in the implementation of MLisp functions which behave differently when chained together (ie. executed one after the other). A good example is ↑K, *kill-to-end-of-line* which appends the text from chained kills to the killbuffer.

To use this technique for a set of commands which are to exhibit a chaining behaviour, first pick a magic number. -84, say. Then each command in this set which is chainable should `(setq this-command -84)`. Then to tell if a command is being chained, it suffices to check to see if `(previous-command)` returns -84.

Did I hear you scream "hack"??

previous-line **↑P**
Move dot to the previous line. ↑N and ↑P attempt to keep dot at the same horizontal position as you move from line to line.

previous-page **ESC-V**

Repositions the current window on the current buffer so that the previous page of the buffer is visible in the window (where a *page* is a group of lines slightly smaller than a window). In other words, it flips you backward a page in the buffer. Its inverse is ↑V. If possible, dot is kept where it is, otherwise it is moved to the middle of the new page.

previous-window **↑XP**

Switches to the window (and associated buffer) that is above the current window.

print **[unbound]**

Print the value of the named variable. This is the command you use when you want to inquire about the setting of some switch or parameter.

process-filter-name **[unbound]**

Returns the name of the filter procedure attached to some buffer.

process-id **[unbound]**

Returns the process id of the process attached to some buffer.

process-output **[unbound]**

(process-output) -- Can only be called by the *on-output-procedure* to procure the output generated by the process whose name is given by *MPX-process*. Returns the output as a string.

process-status **[unbound]**

(process-status "process-name") -- Returns -1 if "process-name" isn't a process, 0 if the process is stopped, and 1 if the process is running.

progn **[unbound]**

(progn expressions...) is an MLisp function that evaluates the expressions and returns the value of the last expression evaluated. *Progn* is roughly equivalent to a compound statement (begin-end block) in more conventional languages and is used where you want to execute several expressions when there is space for only one (eg. the *then* or *else* parts of an *if* expression).

provide-prefix-argument **[unbound]**

(provide-prefix-argument <value> <statement>) provides the prefix argument <value> to the <statement>. For example, the most efficient way to skip forward 5 words is:

```
    (provide-prefix-argument 5 (forward-word))
```

See also the command prefix-argument-loop and the variable prefix-argument.

push-back-character **[unbound]**

Takes the character provided as its argument and causes it to be used as the next character read from the keyboard. It is generally only useful in MLisp functions which read characters from the keyboard, and upon finding one that they don't understand, terminate and behave as though the key had been struck to the EMACS keyboard command interpreter. For example, ITS style incremental search.

put-database-entry **[unbound]**

(put-database-entry dbname key) takes the current buffer and stores it into the named database under the given key.

query-replace-string **ESC-Q**

Replace all occurrences of one string with another, starting at dot and ending at the end of the buffer. EMACS prompts for an old and a new string in the minibuffer (the line at the bottom of the screen). See the section on searching, section 14 page 15 for more information on search strings. For each occurrence of the old string, EMACS requests that the user type in a character to tell it what to do (dot will be positioned just after the found string). The possible replies are:

<space>	Change this occurrence and continue to the next.
n	Don't change this occurrence, but continue to the next
r	Enter a recursive-edit. This allows you to make some local changes, then continue the query-replace-string by typing **↑C**.
!	Change this occurrence and all the rest of the occurrences without bothering to ask.
.	Change this one and stop: don't do any more replaces.
↑G	Don't change this occurrence and stop: don't do any more replaces.
?	(or anything else) Print a short list of the query/replace options.

quietly-read-abbrev-file **[unbound]**

Read in and define abbrevs appearing in a named file. This file should have been written using *write-abbrev-file*. Unlike *read-abbrev-file*, an error message is not printed if the file cannot be found.

quit-process **[unbound]**

(quit-process "process-name") -- Send a quit signal to the process.

quote **[unbound]**

Takes a string and inserts quote characters so that any characters which would have been treated specially by the reqular expression search command will be treated as plain characters. For example, **(quote "a.b")** returns "a\.b".

quote-character **↑Q**

Insert into the buffer the next character typed without interpreting it as a command. This is how you insert funny characters. For example, to insert a **↑L** (form feed or page break character) type **↑Q↑L**. This is the only situation where **↑G** isn't interpreted as an abort character.

re-query-replace-string **[unbound]**

re-query-replace-string is identical to *query-replace-string* except that the search string is a regular expression rather than an uninterpreted sequence of characters. See the section on searching, section 14 page 15 for more information.

re-replace-string **[unbound]**

re-replace-string is identical to *replace-string* except that the search string is a regular expression rather than an uninterpreted sequence of characters. See the section on searching, section 14 page 15 for more information.

re-search-forward **[unbound]**

re-search-forward is identical to *search-forward* except that the search string is a regular expression rather than an uninterpreted sequence of characters. See the section on searching, section 14 page 15 for more information.

re-search-reverse **[unbound]**

re-search-reverse is identical to *search-reverse* except that the search string is a regular expression rather than an uninterpreted sequence of characters. See the section on searching, section 14 page 15 for more information.

read-abbrev-file **[unbound]**

Read in and define abbrevs appearing in a named file. This file should have been written using *write-abbrev-file*. An error message is printed if the file cannot be found.

read-file **↑X↑R**

Prompt for the name of a file; erase the contents of the current buffer; read the file into the buffer and associate the name with the buffer. Dot is set to the beginning of the buffer.

recursion-depth **[unbound]**

Returns the depth of nesting within *recursive-edit*'s. It returns 0 at the outermost level.

recursive-edit **[unbound]**

The *recursive-edit* function is a call on the keyboard read/interpret/execute routine. After *recursive-edit* is called the user can enter commands from the keyboard as usual, except that when he exits EMACS by calling *exit-emacs* (typing ↑C) it actually returns from the call to *recursive-edit*. This function is handy for packages that want to pop into some state, let the user do some editing, then when they're done perform some cleanup and let the user resume. For example, a mail system could use this for message composition.

redraw-display **↑L**

Clear the screen and rewrite it. This is useful if some transmission glitch, or a message from a friend, has messed up the screen.

region-around-match **[unbound]**

Region-around-match sets dot and mark around the region matched by the last search. An argument of *n* puts dot and mark around the *n*'th subpattern matched by '\(' and '\)'. This can then be used in conjuction with *region-to-string* to extract fields matched by a patter. For example, consider the following fragment that extracts user names and host names from mail addresses:

```
(re-search-forward "\\([a-z][a-z]*\\) *@ *\\([a-z][a-z]*\\)")
(region-around-match 1)
(setq username (region-to-string))
(region-around-match 2)
(setq host (region-to-string))
```

Applying this MLisp code to the text "send it to jag@vlsi" would set the variable 'username' to "jag" and 'host' to "vlsi".

region-to-process **[unbound]**

(region-to-process "process-name") -- The region is wrapped up and sent to the process.

region-to-string **[unbound]**

Returns the region between dot and mark as a string. Please be kind to the storage allocator, don't use huge strings.

remove-all-local-bindings **[unbound]**

Perform a remove-local-binding for all possible keys; effectively undoes all local bindings. Mode packages should execute this to initialize the local binding table to a clean state.

remove-binding **[unbound]**

Removes the global binding of the given key. Actually, it just rebinds the key to *illegal-operation.*

remove-local-binding **[unbound]**

Removes the local binding of the given key. The global binding will subsequently be used when interpreting the key. ***Bug:*** there really should be some way of saving the current binding of a key, then restoring it later.

replace-string **ESC-R**

Replace all occurrences of one string for another, starting at dot and ending and the end of the buffer. EMACS prompts for an old and a new string in the minibuffer (the line at the bottom of the screen). Unlike **query-replace-string** EMACS doesn't ask any questions about particular occurrences, it just changes them. Dot will be left after the last changed string. See the section on searching, section 14 page 15 for more information on search strings.

reset-filter **[unbound]**

Removes the filter that had been bound to some process in a buffer.

return-prefix-argument **[unbound]**

`(return-prefix-argument` ***n*****`)`** sets the numeric prefix argument to be used by the next function invocation to *n*. The next function may be either the next function in the normal flow of MLisp execution or the next function invoked from a keystroke. *Return-prefix-argument* is to be used by functions that are to be bound to keys and which are to provide a prefix argument for the next keyboard command.

return-to-monitor **↑_**

Recursivly invokes a new shell, allowing the user to enter normal shell commands and run other programs. Return to EMACS by exiting the shell; ie. by typing **↑D**.

save-excursion **[unbound]**

(save-excursion expressions...) is an MLisp function that evaluates the given expressions and returns the value of the last expression evaluated. It is much like *progn* except that before any expressions are executed dot and the current buffer are "marked" (via the marker mechanism) then after the last expression is executed dot and the current buffer are reset to the marked values. This properly takes into account all movements of dot and insertions and deletions that occur. *Save-excursion* is useful in MLisp functions where you want to go do something somewhere else in this or some other buffer but want to return to the same place when you're done; for example, inserting a tab at the beginning of the current line.

save-restriction **[unbound]**

Save-restriction is only useful to people writing MLisp programs. It is used to save the region restriction for the current buffer (and ***only*** the region restriction) during the execution of some subexpression that presumably uses region restrictions. The value of **`(save-excursion expressions...)`** is the value of the last expression evaluated.

save-window-excursion **[unbound]**
save-window-excursion is identical to *save-excursion* except that it also saves (in a rough sort of way) the state of the windows. That is, **(save-window-excursion expressions...)** saves the current dot, mark, buffer and window state, executes the expressions, restores the saved information and returns the value of the last expression evaluated.

When the window state is saved EMACS remembers which buffers were visible. When it is restored, EMACS makes sure that exactly those buffers are visible. EMACS does *not* save and restore the exact layout of the windows: this is a feature, not a bug.

scroll-one-line-down **ESC-Z**
Repositions the current window on the current buffer so that the line which is currently the second to the last line in the window becomes the last -- effectivly it moves the buffer down one line in the window. **↑Z** is its inverse.

scroll-one-line-up **↑Z**
Repositions the current window on the current buffer so that the line which is currently the second line in the window becomes the first -- effectivly it moves the buffer up one line in the window. ESC-Z is its inverse.

search-forward **↑S**
Prompt for a string and search for a match in the current buffer, moving forwards from dot, stopping at the end of the buffer. Dot is left at the end of the matched string if a match is found, or is unmoved if not. See the section on searching, section 14 page 15 for more information.

search-reverse **↑R**
Prompt for a string and search for a match in the current buffer, moving backwards from dot, stopping at the beginning of the buffer. Dot is left at the beginning of the matched string if a match is found, or is unmoved if not. See the section on searching, section 14 page 15 for more information.

self-insert **[unbound]**
This is tied to those keys which are supposed to self-insert. It is roughly the same as (insert-character (last-key-struck)) with the exception that it doesn't work unless it is bound to a key.

send-string-to-terminal **[unbound]**
(send-string-to-terminal "string") sends the string argumetn out to the terminal with *no* conversion or interpretation. This should only be used for such applications as loading function keys when EMACS starts up. If you screw up the screen, EMACS won't know about it and won't fix it up automatically for you -- you'll have to type **↑L**.

set **[unbound]**
Set the value of some variable internal to EMACS. EMACS will ask for the name of a variable and a value to set it to. The variables control such things as margins, display layout options, the behavior of search commands, and much more. The available variables and switches are described elsewhere. Note that if *set* is used from MLisp the variable name must be a string: (set "left-margin" 77).

set-auto-fill-hook **[unbound]**
set-auto-fill-hook associates a command with the current buffer. When the right margin is passed by the attempt to insert some character the hook procedure for that buffer is invoked. The character that triggered the hook will not have been inserted, but will be inserted immediatly after the hook procedure returns [unless the procedure returns 0]. The hook procedure is responsible for maintaining the position of dot. last-key-struck may be usually used to determine which character triggered the hook. If no hook procedure is associated with a buffer then the old action (break the line and indent) will be taken. This procedure may be used for such things as automatically putting boxes around paragraph comments as they are typed.

set-default **[unbound]**
This commands bears the same relationship to *setq-default* that *set* does to *setq*. It is the command that you use from the keyboard to set the default value of some variable. See the description of *setq-default* for more detailed information.

set-mark **↑@**
Puts the marker for this buffer at the place where dot is now, and leaves it there. As text is inserted or deleted around the mark, the mark will remain in place. Use ↑X↑X to move to the currently marked position.

setq **[unbound]**
Assigns a new value to a variable. Variables may have either string or integer values. (setq i 5) sets i to 5; (setq s (concat "a" "b")) sets s to "ab".

setq-default **[unbound]**
Setq-default is used to set the default value of some variable. It can be a global parameter, a buffer-specific variable or a system variable. It makes no matter, *setq-default* will set the default. *Setq-default* is the command to use from within some MLisp program, like your start up profile (".EMACS_pro"). For example, **(setq-default right-margin 60)** will set the default right margin for newly created buffers to 60. In previous versions of EMACS certain system variables had default versions from which default values were taken. So, to set the default value of *right-margin* one would assign a value to *default-right-margin* -- but no more. Use *setq-default* (or *set-default* instead.

The precise semantics of *setq-default* are:

- If the variable being assigned to has not yet been declared, then declare it as a global variable.
- If it is a global variable (whether or not the declaration was implicit) then assign the value to it just as the *setq* command would have done.
- Otherwise, if the variable is buffer specific then set the default value for the variable. This will be

used in all buffers where the variable hasn't been explicitly assigned a value. Note that if you have a global variable which is eventually declared buffer-specific then the global value becomes the default. The intent of this is that users should be able to put *setq-default*'s in their .emacs–pro's without concerning themselves over whether the variable will eventually be a simple global or buffer-specific.

shell **[unbound]**

The *shell* command is used to either start or reenter a shell process. When the shell command is executed, if a shell process doesn't exist then one is created (running the standard "sh") tied to a buffer named "shell'. In any case, the shell buffer becomes the current one and dot is positioned at the end of it. In that buffer output from the shell and programs run with it will appear. Anything typed into it will get sent to the subprocess when the *return* key is struck. This lets you interact with a shell using EMACS, and all of it's editing capability, as an intermediary. You can scroll backwards over a session, pick up pieces of text from other places and use them as input, edit while watching the execution of some program, and much more...

shrink-window **↑X↑Z**

Makes the current window one line shorter, and the window below (or the one above if there is no window below) one line taller. Can't be used if there is only one window on the screen.

sit-for **[unbound]**

Updates the display and pauses for n/10 seconds. (sit-for 10) waits for one second. This is useful in such things as a Lisp auto-paren balencer.

split-current-window **↑X2**

Enter two-window mode. Actually, it takes the current window and splits it into two windows, dividing the space on the screen equally between the two windows. An arbitrary number of windows can be created -- the only limit is on the amount of space available on the screen, which, *sigh*, is only 24 lines on most terminals available these days (with the notable exception of the Ann Arbor Ambassador which has 60).

start-filtered-process **[unbound]**

(start-filtered-process "command" "buffer-name" "on-output-procedure") -- Does the same thing as start-process except that things are set up so that "on-output-procedure" is automatically called whenever output has been received from this process. This procedure can access the name of the process producing the output by refering to the variable *MPX-process*, and can retrieve the output itself by calling the procedure *process-output*.

The filter procedure must be careful to avoid generating side-effects (eg. *search-forward*). Moreover, if it attempts to go to the terminal for information, output from other processes may be lost.

start-process **[unbound]**
(start-process "command" "buffer-name") -- The home shell is used to start a process executing the command. This process is tied to the buffer "buffer-name" unless it is null in which case the "Command execution" buffer is used. Output from the process is automatically attached to the end of the buffer. Each time this is done, the mark is left at the end of the output (which is the end of the buffer).

start-remembering **↑X(**
All following keystrokes will be remembered by EMACS.

stop-process **[unbound]**
(stop-process "process-name") -- Tell the process to stop by sending it a stop signal. Use *continue-process* to carry on.

stop-remembering **↑X)**
Stops remembering keystrokes, as initiated by **↑X(**. The remembered keystrokes are not forgotten and may be re-executed with **↑XE**.

string-to-char **[unbound]**
Returns the integer value of the first character of its string argument. (string-to-char "0") = '0'.

string-to-process **[unbound]**
(string-to-process "process-name" "string") -- The string is sent to the process.

substr **[unbound]**
(substr str pos n) returns the substring of string *str* starting at position *pos* (numbering from 1) and running for *n* characters. If *pos* is less than 0, then length of the string is added to it; the same is done for *n*. (substr "kzin" 2 2) = "zi"; (substr "blotto.c" -2 2) = ".c".

switch-to-buffer **↑XB**
Prompt for the name of the buffer and associate it with the current window. The old buffer associated with this window merely loses that association: it is not erased or changed in any way. If the new buffer does not exist, it will be created, in contrast with **↑X↑O**.

system-name **[unbound]**
Is an MLisp function that returns the name of the system on which EMACS is being run. This should be the ArpaNet or EtherNet (or whatever) host name of the machine.

temp-use-buffer **[unbound]**

Switch to a named buffer *without* changing window associations. The commands pop-to-buffer and switch-to-buffer both cause a window to be tied to the selected buffer, temp-use-buffer does not. There are a couple of problems that you must beware when using this command: The keyboard command driver insists that the buffer tied to the current window be the current buffer, if it sees a difference then it changes the current buffer to be the one tied to the current window. This means that temp-use-buffer will be ineffective from the keyboard, switch-to-buffer should be used instead. The other problem is that "dot" is really a rather funny concept. There is a value of "dot" associated with each *window*, not with each *buffer*. This is done so that there is a valid interpretation to having the same buffer visible in several windows. There is also a value of "dot" associated with the current buffer. When you switch to a buffer with temp-use-buffer, this "transient dot" is what gets used. So, if you switch to another buffer, then use temp-use-buffer to get back, "dot" will have been set to 1. You can use save-excursion to remember your position.

to-col **[unbound]**

(to-col *n*) is an MLisp function that insert tabs and spaces to move the following character to printing column *n*.

transpose-characters **↑T**

Take the two characters preceding dot and exchange them. One of the most common errors for typists to make is transposing two letters, typing "hte" when "the" is meant. ↑T makes correcting these errors easy, especially if you can develop a "↑T reflex".

undo **[unbound]**

Undoes the effects of the last command typed. Arbitrarily complicated commands may be undone successfully. Only the buffer modifying effects of a command may be undone -- variable assignments, key bindings and similar operations will not be undone. Even 'undo' may be undone, so executing undo twice in a row effectivly does nothing. See the section on undoing, page 39.

undo-boundary **[unbound]**

undo-boundary lays down the boundary between two undoable commands. When commands are undone, a 'command' is considered to be the series of operations between undo boundaries. Normally, they are laid down between keystrokes but MLisp functions may choose to lay down more. See the section on undoing, page 39.

undo-more **[unbound]**

Undoes one more command from what was last undone. **undo-more** must be preceeded by either an **undo** or an **undo-more**. This is usually used by first invoking **undo** to undo a command, then invoking **undo-more** repeatedly to undo more and more commands, until you've retreated to the state you want to be back to. See the section on undoing, page 39.

unlink-file **[unbound]**
(unlink-file *fn*) attempts to unlink (remove) the file named *fn*. It returns true if the unlink failed.

use-abbrev-table **[unbound]**
Sets the current local abbrev table to the one with the given name. Local abbrev tables are buffer specific and are usually set depending on the major mode. Several buffers may have the same local abbrev table. If either the selected abbrev table or the global abbrev table have had some abbrevs defined in them, *abbrev-mode* is turned on for the current buffer.

use-global-map **[unbound]**
(use-global-map "mapname") uses the named map to be used for the global interpretation of all key strokes. *use-local-map* is used to change the local interpretation of key strokes. See the section on keymaps, 15 page 17, for more information.

use-local-map **[unbound]**
(use-local-map "mapname") uses the named map to be used for the local interpretation of all key strokes. *use-global-map* is used to change the global interpretation of key strokes. See the section on keymaps, 15 page 17, for more information.

use-old-buffer **↑X↑O**
Prompt for the name of the buffer and associate it with the current window. The old buffer associated with this window merely loses that association: it is not erased or changed in any way. The buffer must already exist, in contrast with **↑XB**.

use-syntax-table **[unbound]**
Associates the named syntax table with the current buffer. See the description of the modify-syntax-entry command for more information on syntax tables.

users-full-name **[unbound]**
MLisp function that returns the users full name as a string. [Really, it returns the contents of the gecos field of the passwd entry for the current user, which is used on many systems for the users full name.]

users-login-name **[unbound]**
MLisp function that returns the users login name as a string.

visit-file **↑X↑V**
Visit-file asks for the name of a file and switches to a buffer that contains it. The file name is expanded to it's full absolute form (that is, it will start with a '/'). If no buffer contains the file already then EMACS will switch to a new buffer and read the file into it. The name of this new buffer will be just the last component of the file name (everything after the last '/' in the name). If there is already a buffer by that name, and it contains some other file, then EMACS will ask "Enter a new buffer name or <CR> to overwrite the old buffer". For example, if my current directory is "/usr/jag/emacs" and I do a ↑X↑V and give EMACS the file name "../.emacs_pro"then the name of the new buffer will be ".emacs_pro" and the file name will be

"/usr/jag/.emacs_pro". ↑X↑V is the approved way of switching from one file to another within an invocation of EMACS.

while **[unbound]**
(while test expressions...) is an MLisp function that executes the given expressions while the test is true.

widen-region **[unbound]**
The *widen-region* command sets the restriction to encompass the entire buffer. It is usualy used after a *narrow-region* to restore EMACS's attention to the whole buffer.

window-height **[unbound]**
Returns the number of text lines of a window that are visible on the screen.

working-directory **[unbound]**
Returns the pathname of the current working directory.

write-abbrev-file **[unbound]**
Write all defined abbrevs to a named file. This file is suitable for reading back with *read-abbrev-file.*

write-current-file **↑X↑S**
Write the contents of the current buffer to the file whose name is associated with the buffer.

write-file-exit **↑X↑F**
Write all modified buffers to their associated files and if all goes well, EMACS will exit.

write-modified-files **↑X↑M**
Write each modified buffer (as indicated by ↑X↑B) onto the file whose name is associated with the buffer. EMACS will complain if a modified buffer does not have an associated file.

write-named-file **↑X↑W**
Prompt for a name; write the contents of the current buffer to the named file.

yank-buffer **ESC-↑Y**
Take the contents of the buffer whose name is prompted for and insert it at dot in the current buffer. Dot is left after the inserted text.

yank-from-killbuffer ↑Y

Take the contents of the kill buffer and inserts it at dot in the current buffer. Dot is left after the inserted text.

| **[unbound]**

(| e_1 e_2) MLisp function that returns $e_1 | e_2$.

22. Options

This chapter describes (in alpahbetical order) all of the variables which the user may set to configure EMACS to taste.

ask-about-buffer-names

The *ask-about-buffer-names* variable controls what the *visit-file* command does if it detects a collision when constructing a buffer name. If *ask-about-buffer-names* is true (the default) then Emacs will ask for a new buffer name to be given, or for <CR> to be typed which will overwrite the old buffer. If it is false then a buffer name will be synthesized by appending "<*n*>" to the buffer name, for a unique value of *n*. For example, if I *visit-file* "makefile" then the buffer name will be "makefile"; then if I *visit-file* "man/makefile" the buffer name will be "makefile<2>".

backup-by-copying

If true, then when a backup of a file is made (see the section on the backup-before-writing variable) then rather than doing the fancy link/unlink footwork, EMACS copies the original file onto the backup. This preserves all link and owner information & ensures that the files I-number doesn't change (you're crazy if you worry about a files I-number). Backup-by-copying incurs a fairly heafty performance penalty. See the section on the backup-by-copying-when-linked variable for a description of a compromise. (default OFF)

backup-by-copying-when-linked

If true, then when a backup of a file is made (see the section on the backup-before-writing variable) then if the link count of the file is greater than 1, rather than doing the fancy link/unlink footwork, EMACS copies the original file onto the backup. If the link count is 1, then the link/unlink trick is pulled. This preserves link information when it is important, but still manages reasonable performance the rest of the time. See the section on the backup-by-copying variable for a description of a how to have owner & I-number information preserved. (default OFF)

backup-when-writing

If ON EMACS will make a backup of a file just before the first time that it is overwritten. The backup will have the same name as the original, except that the string ".BAK" will be appended; unless the last name in the path has more than 10 characters, in which case it will be truncated to 10 characters. "foo.c" gets backed up on "foo.c.BAK"; "/usr/jag/foo.c" on "/usr/jag/foo.c.BAK"; and "EtherService.c" on "EtherServi.BAK". The backup will only be made the first time that the file is rewritten from within the same invocation of EMACS, so if you write out the file several times the .BAK file will contain the file as it was before EMACS was invoked. The backup is normally made by fancy footwork with links and unlinks, to achieve acceptable performance: when "foo.c" is to be rewritten, EMACS effectivly executes a "mv foo.c foo.c.BAK" and then creates foo.c a write the new copy. The file protection of foo.c is copied from the old foo.c, but old links to the file now point to the .BAK file, and the owner of the new file is the person running EMACS. If you don't like this behaviour, see the switches backup-by-copying and backup-by-copying-when-linked. (default OFF)

buffer-is-modified

Buffer-is-modified is true iff the current buffer has been modified since it was last written out. You may set if OFF (ie. to 0) if you want EMACS to ignore the mods that have been made to this buffer -- it doesn't get you back to the unmodified version, it just tells EMACS not to write it out with the other modified files. EMACS sets buffer-is-modified true any time the buffer is modified.

case-fold-search

If set ON all searches will ignore the case of alphabetics when doing comparisons. (default OFF)

checkpoint-frequency

The number of keystrokes between checkpoints. Every "checkpoint-frequency" keystrokes all buffers which have been modified since they were last checkpointed are written to a file named "*file*.CKP". *File* is the file name associated with the buffer, or if that is null, the name of the buffer. Proper account is taken of the restriction on file names to 14 characters. (default 300)

comment-column

The column at which comments are to start. Used by the language-dependent commenting features through the *move-to-comment-column* command. (default 33)

ctlchar-with-↑

If set ON control characters are printed as ↑C (an '↑' character followed by the upper case alphabetic that corresponds to the control character), otherwise they are printed according to the usual Unix convention ('\' followed by a three digit octal number). (default OFF)

files-should-end-with-newline

Indicates that when a buffer is written to a file, and the buffer doesn't end in a newline, then the user should be asked if they want to have a newline appended. It used to be that this was the default action, but some people objected to the question being asked. (default ON)

global-mode-string

Global-mode-string is a global variable used in the construction of mode lines see section 17, page 19 for more information.

help-on-command-completion-error
If ON EMACS will print a list of possibilities when an ambiguous command is given, otherwise it just rings the bell and waits for you to type more. (default ON)

left-margin
The left margin for automatic text justification. After an automatically generated *newline* the new line will be indented to the left margin.

mode-line-format
mode-line-format is a buffer specific variable used to specify the format of a mode line. See section 17, page 19 for more information.

mode-string
Mode-string is a buffer specific variable used in the construction of mode lines see section 17, page 19 for more information.

needs-checkpointing
A buffer-specific variable which if ON indicates that the buffer should be checkpointed periodically. If it is OFF, then no checkpoints will be done. (default ON)

pop-up-windows
If ON EMACS will try to use some window other than the current one when it spontaneously generates a buffer that it wants you to see or when you visit a file (it may split the current window). If OFF the current window is always used. (default ON)

prefix-argument
Every function invocation is always prefixed by a numeric argument, either explicitly with ↑U*n* or provide-prefix-argument. The value of the variable prefix-argument is the argument prefixed to the invocation of the current MLisp function. For example, if the following function:

```
(defun
    (show-it
        (message (concat "The prefix argument is " prefix-argument))
    )
)
```

were bound to the key ↑A then typing ↑U↑A would cause the message "The prefix argument is 4" to be printed, and ↑U13↑A would print "The prefix argument is 13". See also the commands prefix-argument-loop and provide-prefix-argument.

prefix-argument-provided

True iff the execution of the current function was prefixed by a numeric argument. Use *prefix-argument* to get it's value.

prefix-string

The string that is inserted after an automatic *newline* has been generated in response to going past the right margin. This is generally used by the language-dependent commenting features. (default "")

quick-redisplay

If ON EMACS won't worry so much about the case where you have the same buffer on view in several windows -- it may let the other windows be inaccurate for a short while (but they will eventually be fixed up). Turning this ON speeds up EMACS substantially when the same buffer is on view in several windows. When it is OFF, all windows are always accurate. (default OFF)

replace-case

If ON EMACS will alter the case of strings substituted with *replace-string* or *query-replace-string* to match the case of the original string. For example, replacing "which" by "that" in the string "Which is silly" results in "That is silly"; in the string "the car which is red" results in "the car that is red"; and in the string "WHICH THING?" results in "THAT THING?".

right-margin

The right margin for automatic text justification. If a character is inserted at the end of a line and to the right of the right margin EMACS will automatically insert at the beginning of the preceding word a newline, tabs and spaces to indent to the left margin, and the prefix string. With the right margin set to something like (for eg.) 72 you can type in a document without worrying about when to hit the *return* key, EMACS will automatically do it for you at exactly the right place.

scroll-step

The number of lines by which windows are scrolled if dot moves outside the window. If dot has moved more than *scroll-step* lines outside of the window or *scroll-step* is zero then dot is centered in the window. Otherwise the window is moved up or down *scroll-step* lines. Setting *scroll-step* to 1 will cause the window to scroll by 1 line if you're typing at the end of the window and hit RETURN.

silently-kill-processes

If ON EMACS will kill processes when it exits *without* asking any questions. Normally, if you have processes running when EMACS exits, the question "You have processes on the prowl, should I hunt them down for you" is asked. (default OFF)

stack-trace-on-error
If ON EMACS will write a MLisp stack trace to the "Stack trace" buffer whenever an error is encountered from within an MLisp function (even inside an *error-occured*). This is all there is in the way of a debugging facility. (default OFF)

tab-size
A buffer-specific variable which specifies the number of characters between tab stops. It's not clear that user specifiable tabs are a good idea, since the rest of Unix and most other DEC styled operating systems have the magic number 8 so deeply wired into them. (default 8)

this-command
The meaning of the variable *this-command* is tightly intertwined with the meaning of the function *previous-command.* Look at its documentation for a description of *this-command.*

track-eol-on-↑N-↑P
If ON then ↑N and ↑P will "stick" to the end of a line if they are started there. If OFF ↑N and ↑P will try to stay in the same column as you move up and down even if you started at the end of a line. (default ON)

unlink-checkpoint-files
If ON EMACS will unlink the corresponding checkpoint file after the master copy is written -- this avoids having a lot of .CKP files lying around but it does compromise safety a little. For example, as you're editing a file called "foo.c" EMACS will be periodically be writing a checkpoint file called "foo.c.CKP" that contains all of your recent changes. When you rewrite the file (with ↑X↑F or ↑X↑S for example) if unlink-checkpoint-files is ON then the .CKP file will be unlinked, otherwise it will be left. (default OFF)

visible-bell
If ON EMACS will attempt to use a visible bell, usually a horrendous flashing of the screen, instead of the audible bell, when it is notifying you of some error. This is a more "socially acceptable" technique when people are working in a crowded terminal room. (default OFF)

wrap-long-lines
If ON EMACS will display long lines by "wrapping" their continuation onto the next line (the first line will be terminated with a '\'). If OFF long lines get truncated at the right edge of the screen and a '$' is display to indicate that this has happened. (default OFF)

Unix Emacs Reference Card

SOME NECESSARY NOTATION

	Any ordinary character goes into the buffer (no insert command needed). Commands are all control characters or other characters prefixed by Escape or a control-X. Escape is sometimes called Meta or Altmode in EMACS.
↑	A control character. ↑F means "control F".
ESC·	A two-character command sequence where the first character is Escape. ESC-F means "ESCAPE then F".
ESC·X string	A command designated "by hand". "ESC-x read-file" means: type "Escape", then "x", then "read-file", then <cr>.
dot	EMACS term for cursor position in current buffer.
mark	An invisible set position in the buffer used by region commands.
region	The area of the buffer between the dot and mark.

CHARACTER OPERATIONS

↑B	Move left (Back)
↑F	Move right (Forward)
↑P	Move up (Previous)
↑N	Move down (Next)
↑D	Delete right
↑H or BS or DEL or RUBOUT	Delete left
↑T	Transpose previous 2 characters (ht— -> th—)
↑Q	Literally inserts (quotes) the next character typed (e.g. ↑Q-↑L)
↑U·n	Provide a numeric argument of n to the command that follows (n defaults to 4, eg. try ↑U-↑N and ↑U-↑U-↑F)
↑M or CR	newline
↑J or NL	newline followed by an indent

WORD OPERATIONS

ESC·b	Move left (Back)
ESC·f	Move right (Forward)
ESC·d	Delete word right
ESC·h	Delete word left
ESC·c	Capitalize word
ESC·l	Lowercase word
ESC·u	Uppercase word
ESC·↑	Invert case of word

LINE OPERATIONS

↑A	Move to the beginning of the line
↑E	Move to the end of the line
↑O	Open up a line for typing
↑K	Kill from dot to end of line (↑Y yanks it back at dot)

PARAGRAPH OPERATIONS

ESC·[	Move to beginning of the paragraph
ESC·]	Move to end of the paragraph
ESC·j	Justify the current paragraph

GETTING OUT

↑X·↑S	Save the file being worked on
↑X·↑W	Write the current buffer into a file with a different name
↑X·↑M	Write out all modified files
↑X·↑F	Write out all modified files and exit
↑C or ESC·↑C or ↑X·↑C	Finish by exiting to the shell
↑—	Recursively push (escape) to a new shell

SCREEN AND SCREEN OPERATIONS

↑V	Show next screen page
ESC·V	Show previous screen page
↑L	Redisplay screen
↑Z	Scroll screen up
ESC·Z	Scroll screen down
ESC·!	Move the line dot is on to top of the screen
ESC·,	Move cursor to beginning of window
ESC·.	Move cursor to end of window
↑X·2	Split the current window in two windows (same buffer s each)
↑X·1	Resume single window (using current buffer)
↑X·d	Delete the current window, giving space to window belo
↑X·n	Move cursor to next window
↑X·p	Move cursor to previous window
ESC·↑V	Display the next screen page in the other window
↑X·↑Z	Shrink window
↑X·z	Enlarge window

BUFFER AND FILE OPERATIONS

↑Y	Yank back the last thing killed (kill and delete are differe
↑X·↑V	Get a file into a buffer for editing
↑X·↑R	Read a file into current buffer, erasing old contents
↑X·↑I	Insert file at dot
↑X·↑O	Select a different buffer (it must already exist)
↑X·B	Select a different buffer (it need not pre-exist)
↑X·↑B	Display a list of available buffers
ESC·↑Y	Insert selected buffer at dot
ESC·<	Move to the top of the current buffer
ESC·>	Move to the end of the current buffer

HELP AND HELPER FUNCTIONS

↑G	Abort anything at any time.
ESC·?	Show every command containing string (try ESC-? para)
ESC·X info	Browse through the Emacs manual.
↑X↑U	Undo the effects of previous commands.

SEARCH

↑S	Search forward
↑R	Search backward

REPLACE

ESC·r	Replace one string with another
ESC·q	Query Replace, one string with another

REGION OPERATIONS

↑@	Set the mark
↑X·↑X	Interchange dot and mark (i.e. go to the other end of the
↑W	Kill region (↑Y yanks it back at dot)

MACRO OPERATIONS

↑X·(	Start remembering keystrokes, ie. start defining a k macro
↑X·)	Stop remembering keystrokes, ie. end the definition
↑X·e	Execute remembered keystrokes, ie. execute the k macro

COMPILING (MAKE) OPERATIONS.

↑X·↑E	Execute the "make" (or other) command, saving out buffer
↑X·↑N	Go to the next error in the file
↑X·!	Execute the given command, saving output in a buffer

MAIL

↑X·r	Read mail.
↑X·m	Send mail

Index

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.

›rithm. At this level, profiling counters are simi- to debugging statements whose purpose is to w the number of times a piece of code is exe- ed. Another view of such counters is as boolean ies. One may be interested that a portion of e has executed at all, for exhaustive testing, or :heck that one implementation of an abstraction ıpletely replaces a previous one.

Execution counts are not necessarily propor- ıal to the amount of time required to execute routine or statement. Further, the execution e of a routine will not be the same for all calls on routine. The criteria for establishing execution e must be decided. If a routine implements an traction by invoking other abstractions, the time nt in the routine will not accurately reflect the e required by the abstraction it implements. ıilarly, if an abstraction is implemented by eral routines the time required by the abstrac- ı will be distributed across those routines.

Given the execution time of individual routines, of accounts to each routine the time spent for it the routines it invokes. This accounting is done assembling a *call graph* with nodes that are the tines of the program and directed arcs that resent calls from call sites to routines. We dis- ;uish among three different call graphs for a pro- m. The *complete call graph* incorporates all rou- ›s and all potential arcs, including arcs that resent calls to functional parameters or func- ıal variables. This graph contains the other two phs as subgraphs. The *static call graph* includes routines and all possible arcs that are not calls 'unctional parameters or variables. The *dynamic l graph* includes only those routines and arcs versed by the profiled execution of the program. s graph need not include all routines, nor need it lude all potential arcs between the routines it 'ers. It may, however, include arcs to functional ·ameters or variables that the static call graph y omit. The static call graph can be determined m the (static) program text. The dynamic call ıph is determined only by profiling an execution the program. The complete call graph for a nolithic program could be determined by data ᵣ analysis techniques. The complete call graph programs that change during execution, by difying themselves or dynamically loading or :rlaying code, may never be determinable. Both : static call graph and the dynamic call graph are :d by **gprof**, but it does not search for the com- te call graph.

Gathering Profile Data

Routine calls or statement executions can be ıasured by having a compiler augment the code strategic points. The additions can be inline :rements to counters [Knuth71] [Satterthwaite72] ›y79] or calls to monitoring routines [Unix]. The ınter increment overhead is low, and is suitable · profiling statements. A call of the monitoring ıtine has an overhead comparable with a call of a ;ular routine, and is therefore only suited to ɔfiling 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:

$$T_r = S_r + \sum_{r \text{ CALLS } e} T_e \times \frac{C_e^r}{C_e}$$

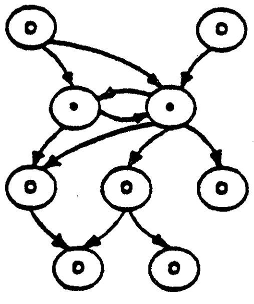
where r CALLS e is a relation showing all routines e called by a routine r. This relation is easily available from the call graph.

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].

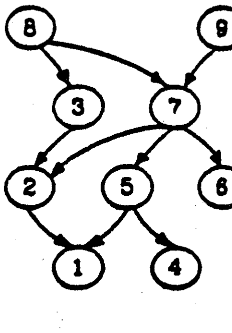
Time propagation within strongly connected components is a problem. For example, a self-recursive routine (a trivial cycle in the call graph) is accountable for all the time it uses in all its recursive instantiations. In our scheme, this time should be shared among its call graph parents. The arcs from 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.

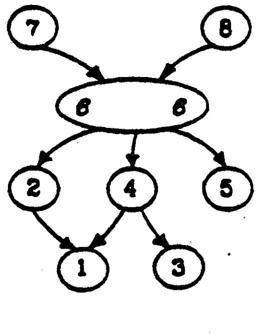
Since the technique described above only collects the dynamic call graph, and the program typically does not call every routine on each execution, different executions can introduce different cycles in the dynamic call graph. Since cycles often have a significant effect on time propagation, it is desirable 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.

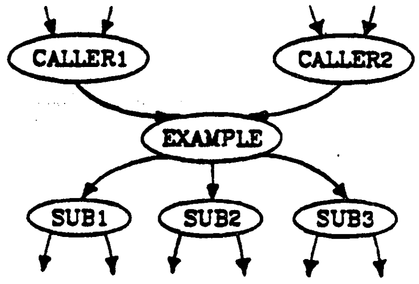
5.2. The Call Graph Profile

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 a
the entries from the flat profile, augmented by t
time propagated to each routine from its desce
dants. This profile is sorted by the sum of the tir
for the routine itself plus the time inherited frc
its descendants. The profile shows which of t
higher level routines spend large portions of t
total execution time in the routines that they ca
For each routine, we show the amount of tir
passed by each child to the routine, which includ
time for the child itself and for the descendants
the child (and thus the descendants of the routin
We also show the percentage these times represe
of the total time accounted to the child. Similar
the parents of each routine are listed, along wi
time, and percentage of total routine time, pr
pagated to each one.

Cycles are handled as single entities. The cy
as a whole is shown as though it were a single ro
tine, except that members of the cycle are listed
place of the children. Although the number of ca
of each member from within the cycle are show
they do not affect time propagation. When a child
a member of a cycle, the time shown is t
appropriate fraction of the time for the whole cyc
Self-recursive routines have their calls broken do
into calls from the outside and self-recursive ca
Only the outside calls affect the propagation
time.

The following example is a typical fragment c
call graph.



The entry in the call graph profile listing for t
example is shown in Figure 4.

The entry is for routine EXAMPLE, which has
Caller routines as its parents, and the Sub routi
as its children. The reader should keep in m
that all information is given *with respect to EX*
PLE. The index in the first column shows that EX
PLE is the second entry in the profile listing.
EXAMPLE routine is called ten times, four times
CALLER1, and six times by CALLER2. Conseque
40% of EXAMPLE's time is propagated to CALLER1,
60% of EXAMPLE's time is propagated to CALLE
The self and descendant fields of the parents s
the amount of self and descendant time EXAM
propagates to them (but not the time used by
parents directly). Note that EXAMPLE calls it
recursively four times. The routine EXAMPLE c
routine SUB1 twenty times, SUB2 once, and ne
calls SUB3. Since SUB2 is called a total of five tin
20% of its self and descendant time is propagate
EXAMPLE's descendant time field. Because SUB1

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.

ember of *cycle 1*, the self and descendant times ıd call count fraction are those for the cycle as a ıole. Since cycle 1 is called a total of forty times .ot counting calls among members of the cycle), it ·opagates 50% of the cycle's self and descendant me to EXAMPLE's descendant time field. Finally ıch name is followed by an index that shows where ı the listing to find the entry for that routine.

Using the Profiles

The profiler is a useful tool for improving a set routines that implement an abstraction. It can : helpful in identifying poorly coded routines, and evaluating the new algorithms and code that :place them. Taking full advantage of the profiler :quires a careful examination of the call graph ·ofile, and a thorough knowledge of the abstrac- ons underlying the program.

The easiest optimization that can be performed a small change to a control construct or data ructure that improves the running time of the rogram. An obvious starting point is a routine that called many times. For example, suppose an out- ut routine is the only parent of a routine that for- ıats the data. If this format routine is expanded iline in the output routine, the overhead of a func- on call and return can be saved for each datum ıat needs to be formatted.

The drawback to inline expansion is that the ata abstractions in the program may become less arameterized, hence less clearly defined. The rofiling will also become less useful since the loss ! routines will make its output more granular. For xample, if the symbol table functions "lookup", insert", and "delete" are all merged into a single arameterized routine, it will be impossible to etermine the costs of any one of these individual ınctions from the profile.

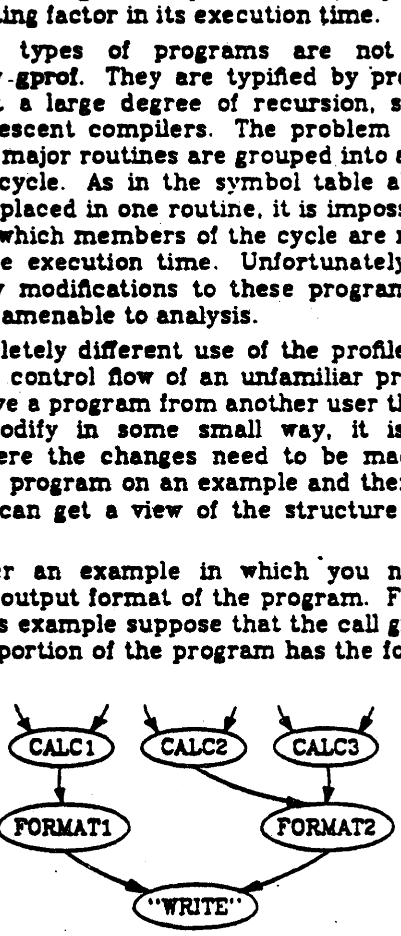
Further potential for optimization lies in rou- nes that implement data abstractions whose total xecution time is long. For example, a lookup rou- ne might be called only a few times, but use an ıefficient linear search algorithm, that might be eplaced with a binary search. Alternately, the iscovery that a rehashing function is being called xcessively, can lead to a different hash function or larger hash table. If the data abstraction function annot easily be speeded up, it may be advanta- eous to cache its results, and eliminate the need to erun it for identical inputs. These and other ideas or program improvement are discussed in [Bent- ey81].

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.

A 4.2bsd Interprocess Communication Primer
DRAFT of July 27, 1983

Samuel J. Leffler

Robert S. Fabry

William N. Joy

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
(415) 642-7780

ABSTRACT

This document provides an introduction to the interprocess communication facilities included in the 4.2bsd release of the VAX* UNIX** system.

It discusses the overall model for interprocess communication and introduces the interprocess communication primitives which have been added to the system. The majority of the document considers the use of these primitives in developing applications. The reader is expected to be familiar with the C programming language as all examples are written in C.

* DEC and VAX are trademarks of Digital Equipment Corporation.
** UNIX is a Trademark of Bell Laboratories.

1. INTRODUCTION

One of the most important parts of 4.2bsd is the interprocess communication facilities. These facilities are the result of more than two years of discussion and research. The facilities provided in 4.2bsd incorporate many of the ideas from current research, while trying to maintain the UNIX philosophy of simplicity and conciseness. It is hoped that the interprocess communication facilities included in 4.2bsd will establish a standard for UNIX. From the response to the design, it appears many organizations carrying out work with UNIX are adopting it.

UNIX has previously been very weak in the area of interprocess communication. Prior to the 4.2bsd facilities, the only standard mechanism which allowed two processes to communicate were pipes (the mpx files which were part of Version 7 were experimental). Unfortunately, pipes are very restrictive in that the two communicating processes must be related through a common ancestor. Further, the semantics of pipes makes them almost impossible to maintain in a distributed environment.

Earlier attempts at extending the ipc facilities of UNIX have met with mixed reaction. The majority of the problems have been related to the fact these facilities have been tied to the UNIX file system; either through naming, or implementation. Consequently, the ipc facilities provided in 4.2bsd have been designed as a totally independent subsystem. The 4.2bsd ipc allows processes to rendezvous in many ways. Processes may rendezvous through a UNIX file system-like name space (a space where all names are path names) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, the communication facilities have been extended to included more than the simple byte stream provided by a pipe-like entity. These extensions have resulted in a completely new part of the system which users will need time to familiarize themselves with. It is likely that as more use is made of these facilities they will be refined; only time will tell.

The remainder of this document is organized in four sections. Section 2 introduces the new system calls and the basic model of communication. Section 3 describes some of the supporting library routines users may find useful in constructing distributed applications. Section 4 is concerned with the client/server model used in developing applications and includes examples of the two major types of servers. Section 5 delves into advanced topics which sophisticated users are likely to encounter when using the ipc facilities.

2. BASICS

The basic building block for communication is the *socket.* A socket is an endpoint of communication to which a name may be *bound.* Each socket in use has a *type* and one or more associated processes. Sockets exist within *communication domains.* A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the UNIX communication domain sockets are named with UNIX path names; e.g. a socket may be named "/dev/foo". Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The 4.2bsd ipc supports two separate communication domains: the UNIX domain, and the Internet domain is used by processes which communicate using the the DARPA standard communication protocols. The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket "operating" in the UNIX domain sees a subset of the possible error conditions which are possible when operating in the Internet domain.

2.1. Socket types

Sockets are typed according to the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Three types of sockets currently are available to a user. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes*.

A *datagram* socket supports bidirectional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets is considered in section 5.

Two potential socket types which have interesting properties are the *sequenced packet* socket and the *reliably delivered message* socket. A sequenced packet socket is identical to a stream socket with the exception that record boundaries are preserved. This interface is very similar to that provided by the Xerox NS Sequenced Packet protocol. The reliably delivered message socket has similar properties to a datagram socket, but with reliable delivery. While these two socket types have been loosely defined, they are currently unimplemented in 4.2bsd. As such, in this document we will concern ourselves only with the three socket types for which support exists.

* In the UNIX domain, in fact, the semantics are identical and, as one might expect, pipes have been implemented internally as simply a pair of connected stream sockets.

2.2. Socket creation

To create a socket the *socket* system call is used:

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

This call requests that the system create a socket in the specified *domain* and of the specified *type*. A particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type. The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets. The domain is specified as one of the manifest constants defined in the file <*sys/socket.h*>. For the UNIX domain the constant is AF_UNIX*; for the Internet domain AF_INET. The socket types are also defined in this file and one of SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW must be specified. To create a stream socket in the Internet domain the following call might be used:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for on-machine use a sample call might be:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

To obtain a particular protocol one selects the protocol number, as defined within the communication domain. For the Internet domain the available protocols are defined in <*netinet/in.h*> or, better yet, one may use one of the library routines discussed in section 3, such as *getprotobyname*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("tcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

2.3. Binding names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. The *bind* call is used to assign a name to a socket:

```
bind(s, name, namelen);
```

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties which comprise the "domain"). In the UNIX domain names are path names while in the Internet domain names contain an Internet address and port number. If one wanted to bind the name "/dev/foo" to a UNIX domain socket, the following would be used:

* The manifest constants are named AF_whatever as they indicate the "address format" to use in interpreting names.

```
bind(s, "/dev/foo", sizeof ("/dev/foo") − 1);
```

(Note how the null byte in the name is not counted as part of the name.) In binding an Internet address things become more complicated. The actual call is simple,

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
bind(s, &sin, sizeof (sin));
```

but the selection of what to place in the address *sin* requires some discussion. We will come back to the problem of formulating Internet addresses in section 3 when the library routines used in name resolution are discussed.

2.4. Connection establishment

With a bound socket it is possible to rendezvous with an unrelated process. This operation is usually asymmetric with one process a "client" and the other a "server". The client requests services from the server by initiating a "connection" to the server's socket. The server, when willing to offer its advertised services, passively "listens" on its socket. On the client side the *connect* call is used to initiate a connection. Using the UNIX domain, this might appear as,

```
connect(s, "server-name", sizeof ("server-name"));
```

while in the Internet domain,

```
struct sockaddr_in server;
connect(s, &server, sizeof (server));
```

If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket; c.f. section 5.4. An error is returned when the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin.

Many errors can be returned when a connection attempt fails. The most common are:

ETIMEDOUT
: After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED
: The host refused service for some reason. When connecting to a host running 4.2bsd this is usually due to a server process not being present at the requested name.

ENETDOWN or EHOSTDOWN
: These operational errors are returned based on status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH
: These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down. In these cases the system is conservative and indicates the entire network is unreachable.

For the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
    listen(s, 5);
```

The second parameter to the *listen* call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages which comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may *accept* a connection:

```
    fromlen = sizeof (from);
    snew = accept(s, &from, &fromlen);
```

A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be zero.

Accept normally blocks. That is, the call to accept will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or not block on the accept call there are alternatives; they will be considered in section 5.

2.5. Data transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect, in this case, then the normal *read* and *write* system calls are useable,

```
    write(s, buf, sizeof (buf));
    read(s, buf, sizeof (buf));
```

In addition to *read* and *write*, the new calls *send* and *recv* may be used:

```
    send(s, buf, sizeof (buf), flags);
    recv(s, buf, sizeof (buf), flags);
```

While *send* and *recv* are virtually identical to *read* and *write*, the extra *flags* argument is important. The flags may be specified as a non-zero value if one or more of the following is required:

SOF_OOB	send/receive out of band data
SOF_PREVIEW	look at data without reading
SOF_DONTROUTE	send data without routing packets

Out of band data is a notion specific to stream sockets, and one which we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When SOF_PREVIEW is specified with a *recv* call, any data present is returned to the user, but treated as still "unread". That is, the next *read* or *recv* call applied to the socket will return the data previously previewed.

2.6. Discarding sockets

Once a socket is no longer of interest, it may be discarded by applying a *close* to the descriptor,

```
close(s);
```

If data is associated with a socket which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it may perform a *shutdown* on the socket prior to closing it. This call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received. Applying shutdown to a socket causes any data queued to be immediately discarded.

2.7. Connectionless sockets

To this point we have been concerned mostly with sockets which follow a connection oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before, and each should have a name bound to it in order that the recipient of a message may identify the sender. To send data, the *sendto* primitive is used,

```
sendto(s, buf, buflen, flags, &to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the intended recipient of the message. When using an unreliable datagram interface, it is unlikely any errors will be reported to the sender. Where information is present locally to recognize a message which may never be delivered (for instance when a network is unreachable), the call will return −1 and the global value *errno* will contain an error number.

To receive messages on an unconnected datagram socket, the *recvfrom* primitive is provided:

```
recvfrom(s, buf, buflen, flags, &from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer.

In addition to the two calls mentioned above, datagram sockets may also use the *connect* call to associate a socket with a specific address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket (i.e. no multicasting). Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket where a connect request initiates establishment of an end to end connection). Other of the less important details of datagram sockets are described in section 5.

2.8. Input/Output multiplexing

One last facility often used in developing applications is the ability to multiplex i/o requests among multiple sockets and/or files. This is done using the *select* call:

```
select(nfds, &readfds, &writefds, &execptfds, &timeout);
```

Select takes as arguments three bit masks, one for the set of file descriptors for which the caller wishes to be able to read data on, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending. Bit masks are created by or-ing bits of the form "1 << fd". That is, a descriptor *fd* is selected if a 1 is present in the *fd*'th bit of the mask. The parameter *nfds* specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) specified in a mask.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If *timeout* is set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely*. *Select* normally returns the number of file descriptors selected. If the *select* call returns due to the timeout expiring, then a value of −1 is returned along with the error number EINTR.

Select provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in section 5.

* To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

3. NETWORK LIBRARY ROUTINES

The discussion in section 2 indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task a number of routines have been added to the standard C run-time library. In this section we will consider the new routines provided to manipulate network addresses. While the 4.2bsd networking facilities support only the DARPA standard Internet protocols, these routines have been designed with flexibility in mind. As more communication protocols become available, we hope the same user interface will be maintained in accessing network-related address data bases. The only difference should be the values returned to the user. Since these values are normally supplied the system, users should not need to be directly aware of the communication protocol and/or naming conventions in use.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; e.g. "the *login server* on host monet". This name, and the name of the peer host, must then be translated into network *addresses* which are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings is likely to vary between network architectures. For instance, it is desirable for a network to not require hosts be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for: mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file <*netdb.h*> must be included when using any of these routines.

3.1. Host names

A host name to address mapping is represented by the *hostent* structure:

```
struct  hostent {
        char      *h_name;        /* official name of host */
        char      **h_aliases;    /* alias list */
        int       h_addrtype;     /* host address type */
        int       h_length;       /* length of address */
        char      *h_addr;        /* address */
};
```

The official name of the host and its public aliases are returned, along with a variable length address and address type. The routine *gethostbyname*(3N) takes a host name and returns a *hostent* structure, while the routine *gethostbyaddr*(3N) maps host addresses into a *hostent* structure. It is possible for a host to have many addresses, all having the same name. *Gethostybyname* returns the first matching entry in the data base file */etc/hosts*; if this is unsuitable, the lower level routine *gethostent*(3N) may be used. For example, to obtain a *hostent* structure for a host on a particular network the following routine might be used (for simplicity, only Internet addresses are considered):

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
struct hostent *
gethostbynameandnet(name, net)
        char *name;
        int net;
{
        register struct hostent *hp;
        register char **cp;

        sethostent(0);
        while ((hp = gethostent()) != NULL) {
                if (hp->h_addrtype != AF_INET)
                        continue;
                if (strcmp(name, hp->h_name)) {
                        for (cp = hp->h_aliases; cp && *cp != NULL; cp++)
                                if (strcmp(name, *cp) == 0)
                                goto found;
                        continue;
                }
        found:
                if (in_netof(*(struct in_addr *)hp->h_addr)) == net)
                        break;
        }
        endhostent(0);
        return (hp);
}
```

(*in_netof*(3N) is a standard routine which returns the network portion of an Internet address.)

3.2. Network names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct  netent {
        char        *n_name;          /* official name of net */
        char        **n_aliases;      /* alias list */
        int         n_addrtype;       /* net address type */
        int         n_net;            /* network # */
};
```

The routines *getnetbyname*(3N), *getnetbynumber*(3N), and *getnetent*(3N) are the network counterparts to the host routines described above.

3.3. Protocol names

For protocols the *protoent* structure defines the protocol-name mapping used with the routines *getprotobyname*(3N), *getprotobynumber*(3N), and *getprotoent*(3N):

```
struct  protoent {
        char        *p_name;            /* official protocol name */
        char        **p_aliases;        /* alias list */
        int         p_proto;            /* protocol # */
};
```

3.4. Service names

Information regarding services is a bit more complicated. A service is expected to reside at a specific "port" and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports or support multiple protocols. If either of these occurs, the higher level library routines will have to be bypassed in favor of homegrown routines similar in spirit to the "gethostbynameandnet" routine described above. A service mapping is described by the *servent* structure,

```
struct  servent {
        char        *s_name;            /* official service name */
        char        **s_aliases;        /* alias list */
        int         s_port;             /* port # */
        char        *s_proto;           /* protocol to use */
};
```

The routine *getservbyname*(3N) maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

```
sp = getservbyname("telnet", (char *)0);
```

returns the service specification for a telnet server using any protocol, while the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routines *getservbyport*(3N) and *getservent*(3N) are also provided. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; an optional protocol name may be specified to qualify lookups.

3.5. Miscellaneous

With the support routines described above, an application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in Figure 1. (This example will be considered in more detail in section 4.)

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be worthwhile. Perhaps when the system is adapted to different network architectures the utilities will be reorganized more cleanly.

Aside from the address-related data base routines, there are several other routines available in the run-time library which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. Table 1 summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On a VAX, or machine with similar architecture, this is usually reversed. Consequently, programs are sometimes required to byte swap quantities. The

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
...
main(argc, argv)
        char *argv[];
{
        struct sockaddr_in sin;
        struct servent *sp;
        struct hostent *hp;
        int s;
        ...
        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service\n");
                exit(1);
        }
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }
        bzero((char *)&sin, sizeof (sin));
        bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
        sin.sin_family = hp->h_addrtype;
        sin.sin_port = sp->s_port;
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
        ...
        if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }
        ...
}
```

Figure 1. Remote login client code.

Call	Synopsis
bcmp(s1, s2, n)	compare byte-strings; 0 if same, not 0 otherwise
bcopy(s1, s2, n)	copy n bytes from s1 to s2
bzero(base, n)	zero-fill n bytes starting at base
htonl(val)	convert 32-bit quantity from host to network byte order
htons(val)	convert 16-bit quantity from host to network byte order
ntohl(val)	convert 32-bit quantity from network to host byte order
ntohs(val)	convert 16-bit quantity from network to host byte order

Table 1. C run-time routines.

library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
	printf("port number %d\n", ntohs(sp->s_port));
```

On machines other than the VAX these routines are defined as null macros.

4. CLIENT/SERVER MODEL

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server which has been examined in section 2. In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

Client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a "client process" and a "server process". We will first consider the properties of server processes, then client processes.

A server process normally listens at a well know address for service requests. Alternative schemes which use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. The Xerox Courier protocol uses the latter scheme. When using Courier, a Courier client process contacts a Courier server at the remote host and identifies the service it requires. The Courier server process then creates the appropriate server process based on a data base and "splices" the client and server together, voiding its part in the transaction. This scheme is attractive in that the Courier server process may provide a single contact point for all services, as well as carrying out the initial steps in authentication. However, while this is an attractive possibility for standardizing access to services, it does introduce a certain amount of overhead due to the intermediate process involved. Implementations which provide this type of service within the system can minimize the cost of client server rendezvous. The *portal* notion described in the "4.2BSD System Manual" embodies many of the ideas found in Courier, with the rendezvous mechanism implemented internal to the system.

4.1. Servers

In 4.2bsd most servers are accessed at well known Internet addresses or UNIX domain names. When a server is started at boot time it advertises it services by listening at a well know location. For example, the remote login server's main loop is of the form shown in Figure 2.

The first step taken by the server is look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service\n");
        exit(1);
}
```

This definition is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

Step two is to disassociate the server from the controlling terminal of its invoker. This is important as the server will likely not want to receive signals delivered to the process group of the controlling terminal.

```
main(argc, argv)
        int argc;
        char **argv;
{
        int f;
        struct sockaddr_in from;
        struct servent *sp;

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogind: tcp/login: unknown service\n");
                exit(1);
        }
        ...
#ifndef DEBUG
        <<disassociate server from controlling terminal>>
#endif
        ...
        sin.sin_port = sp->s_port;
        ...
        f = socket(AF_INET, SOCK_STREAM, 0);
        ...
        if (bind(f, (caddr_t)&sin, sizeof (sin)) < 0) {
                ...
        }
        ...
        listen(f, 5);
        for (;;) {
                int g, len = sizeof (from);

                g = accept(f, &from, &len);
                if (g < 0) {
                        if (errno != EINTR)
                                perror("rlogind: accept");
                        continue;
                }
                if (fork() == 0) {
                        close(f);
                        doit(g, &from);
                }
                close(g);
        }
}
```

Figure 2. Remote login server.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its expected location. The main body of the loop is fairly simple:

```
    for (;;) {
            int g, len = sizeof (from);

            g = accept(f, &from, &len);
            if (g < 0) {
                    if (errno != EINTR)
                            perror("rlogind: accept");
                    continue;
            }
            if (fork() == 0) {
                    close(f);
                    doit(g, &from);
            }
            close(g);
    }
```

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in section 5). Therefore, the return value from *accept* is checked to insure a connection has actually been established. With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queueing connection requests is closed in the child, while the socket created as a result of the accept is closed in the parent. The address of the client is also handed the *doit* routine because it requires it in authenticating clients.

4.2. Clients

The client side of the remote login service was shown earlier in Figure 1. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client remote login process. As in the server process the first step is to locate the service definition for a remote login:

```
    sp = getservbyname("login", "tcp");
    if (sp == NULL) {
          fprintf(stderr, "rlogin: tcp/login: unknown service\n");
          exit(1);
    }
```

Next the destination host is looked up with a *gethostbyname* call:

```
    hp = gethostbyname(argv[1]);
    if (hp == NULL) {
          fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
          exit(2);
    }
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login process resides:

```
    bzero((char *)&sin, sizeof (sin));
    bcopy(hp->h_addr, (char *)sin.sin_addr, hp->h_length);
    sin.sin_family = hp->h_addrtype;
    sin.sin_port = sp->s_port;
```

A socket is created, and a connection initiated.

```
    s = socket(hp->h_addrtype, SOCK_STREAM, 0);
    if (s < 0) {
        perror("rlogin: socket");
        exit(3);
    }
     ...
    if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
        perror("rlogin: connect");
        exit(4);
    }
```

The details of the remote login protocol will not be considered here.

4.3. Connectionless servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the "rwho" service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the rwho server may find out the current status of a machine with the *ruptime*(1) program. The output generated is illustrated in Figure 3.

```
arpa       up   9:45,       5 users, load  1.15,  1.39,  1.31
cad        up   2+12:04,    8 users, load  4.67,  5.13,  4.59
calder     up   10:10,      0 users, load  0.27,  0.15,  0.14
dali       up   2+06:28,    9 users, load  1.04,  1.20,  1.65
degas      up   25+09:48,   0 users, load  1.49,  1.43,  1.41
ear        up   5+00:05,    0 users, load  1.51,  1.54,  1.56
ernie    down   0:24
esvax    down   17:04
ingres   down   0:26
kim        up   3+09:16,    8 users, load  2.03,  2.46,  3.11
matisse    up   3+06:18,    0 users, load  0.03,  0.03,  0.05
medea      up   3+09:39,    2 users, load  0.35,  0.37,  0.50
merlin   down   19+15:37
miro       up   1+07:20,    7 users, load  4.59,  3.28,  2.12
monet      up   1+00:43,    2 users, load  0.22,  0.09,  0.07
oz       down   16:09
statvax    up   2+15:57,    3 users, load  1.52,  1.81,  1.86
ucbvax     up   9:34,       2 users, load  6.08,  5.16,  3.28
```

Figure 3. ruptime output.

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

The rwho server, in a simplified form, is pictured in Figure 4. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to insure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

```
main()
{
        ...
        sp = getservbyname("who", "udp");
        net = getnetbyname("localnet");
        sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
        sin.sin_port = sp->s_port;
        ...
        s = socket(AF_INET, SOCK_DGRAM, 0);
        ...
        bind(s, &sin, sizeof (sin));
        ...
        sigset(SIGALRM, onalrm);
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);

                cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0, &from, &len);
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                perror("rwhod: recv");
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                        fprintf(stderr, "rwhod: %d: bad from port\n",
                                ntohs(from.sin_port));
                        continue;
                }
                ...
                if (!verify(wd.wd_hostname)) {
                        fprintf(stderr, "rwhod: malformed host name from %x\n",
                                ntohl(from.sin_addr.s_addr));
                        continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
                whod = open(path, FWRONLY|FCREATE|FTRUNCATE, 0666);
                ...
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

Figure 4. rwho server.

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet does, however, indicates some problems with the current protocol.

Status information is broadcast on the local network. For networks which do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status received). This, unfortunately, requires some bootstrapping information, as a server started up on a quiet network will have no known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information*.

The second problem with the current scheme is that the rwho process services only a single local network, and this network is found by reading a file. It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. 4.2bsd attempts to isolate host-specific information from applications by providing system calls which return the necessary information†. Unfortunately, no straightforward mechanism currently exists for finding the collection of networks to which a host is directly connected. Thus the rwho server performs a lookup in a file to find its local network. A better, though still unsatisfactory, scheme used by the routing process is to interrogate the system data structures to locate those directly connected networks. A mechanism to acquire this information from the system would be a useful addition.

* One must, however, be concerned about "loops". That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, exchange of information.
† An example of such a system call is the *gethostname*(2) call which returns the host's "official" name.

5. ADVANCED TOPICS

A number of facilities have yet to be discussed. For most users of the ipc the mechanisms already described will suffice in constructing distributed applications. However, others will find need to utilize some of the features which we consider in this section.

5.1. Out of band data

The stream socket abstraction includes the notion of "out of band" data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data along with the SIGURG signal. In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals from between client and server processes. When a signal is expected to flush any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

The stream abstraction defines that the out of band data facilities must support the reliable delivery of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (i.e. the urgent data is delivered in sequence with the normal data) the system extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data.

To send an out of band message the SOF_OOB flag is supplied to a *send* or *sendto* calls, while to receive out of band data SOF_OOB should be indicated when performing a *recvfrom* or *recv* call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK ioctl is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 5.

5.2. Signals and process groups

Due to the existence of the SIGURG and SIGIO signals each socket has an associated process group (just as is done for terminals). This process group is initialized to the process group of its creator, but may be redefined at a later time with the SIOCSPGRP ioctl:

```
oob()
{
        int out = 1+1;
        char waste[BUFSIZ], mark;

        signal(SIGURG, oob);
        /* flush local terminal input and output */
        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof (waste));
        }
        recv(rem, &mark, 1, SOF_OOB);
        ...
}
```

Figure 5. Flushing terminal i/o on receipt of out of band data.

```
ioctl(s, SIOCSPGRP, &pgrp);
```

A similar ioctl, SIOCGPGRP, is available for determining the current process group of a socket.

5.3. Pseudo terminals

Many programs will not function properly without a terminal for standard input and output. Since a socket is not a terminal, it is often necessary to have a process communicating over the network do so through a *pseudo terminal.* A pseudo terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo terminal is supplied as input to a process reading from the master side. Data written on the master side is given the slave as input. In this way, the process manipulating the master side of the pseudo terminal has control over the information read and written on the slave side. The remote login server uses pseudo terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends an interrupt or quit signal to a process executing on a remote machine, the client login program traps the signal, sends an out of band message to the server process who then uses the signal number, sent as the data value in the out of band message, to perform a *killpg*(2) on the appropriate process group.

5.4. Internet address binding

Binding addresses to sockets in the Internet domain can be fairly complex. Communicating processes are bound by an *association.* An association is composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each Internet protocol. Associations are always unique. That is, there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples.

The bind system call allows a process to specify half of an association, <local address, local port>, while the connect and accept primitives are used to complete a socket's association. Since the association is created in two steps the association uniqueness requirement indicated above could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding the notion of a "wildcard" address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in <netinet/in.h>), the system interprets the address as "any valid address". For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and addressed to any of the possible addresses assigned a host. For example, if a host is on a networks 46 and 10 and a socket is bound as above, then an accept call is performed, the process will be able to accept connection requests which arrive either from network 46 or network 10.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. For example:

```
sin.sin_addr.s_addr = MYADDRESS;
sin.sin_port = 0;
bind(s, (char *)&sin, sizeof (sin));
```

The system selects the port number based on two criteria. The first is that ports numbered 0 through 1023 are reserved for privileged users (i.e. the super user). The second is that the port number is not currently bound to some other socket. In order to find a free port number in the privileged range the following code is used by the remote shell server:

```
struct sockaddr_in sin;
...
lport = IPPORT_RESERVED − 1;
sin.sin_addr.s_addr = INADDR_ANY;
...
for (;;) {
     sin.sin_port = htons((u_short)lport);
     if (bind(s, (caddr_t)&sin, sizeof (sin)) >= 0)
          break;
     if (errno != EADDRINUSE && errno != EADDRNOTAVAIL) {
          perror("socket");
          break;
     }
     lport--;
     if (lport == IPPORT_RESERVED/2) {
          fprintf(stderr, "socket: All ports in use\n");
          break;
     }
}
```

The restriction on allocating ports was done to allow processes executing in a "secure" environment to perform authentication based on the originating address and port number.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. This is due to associations being created in a two step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket were around. To override the default port selection algorithm then an option call must be performed prior to address binding:

```
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)0, 0);
bind(s, (char *)&sin, sizeof (sin));
```

With the above call, local addresses may be bound which are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port (if an association already exists, the error EADDRINUSE is returned).

Local address binding by the system is currently done somewhat haphazardly when a host is on multiple networks. Logically, one would expect the system to bind the local address associated with the network through which a peer was communicating. For instance, if the local host is connected to networks 46 and 10 and the foreign host is on network 32, and traffic from network 32 were arriving via network 10, the local address to be bound would be the host's address on network 10, not network 46. This unfortunately, is not always the case. For reasons too complicated to discuss here, the local address bound may be appear to be chosen at random. This property of local address binding will normally be invisible to users unless the foreign host does not understand how to reach the address selected*.

* For example, if network 46 were unknown to the host on network 32, and the local address were bound to that located on network 46, then even though a route between the two hosts existed through network 10, a connection would fail.

5.5. Broadcasting and datagram sockets

By using a datagram socket it is possible to send broadcast packets on many networks supported by the system (the network itself must support the notion of broadcasting; the system provides no broadcast simulation in software). Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to the super user.

To send a broadcast message, an Internet datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Then the message should be addressed as:

```
dst.sin_family = AF_INET;
dst.sin_addr.s_addr = INADDR_ANY;
dst.sin_port = DESTPORT;
```

and, finally, a sendto call may be used:

```
sendto(s, buf, buflen, 0, &dst, sizeof (dst));
```

Received broadcast messages contain the senders address and port (datagram sockets are anchored before a message is allowed to go out).

5.6. Signals

Two new signals have been added to the system which may be used in conjunction with the interprocess communication facilities. The SIGURG signal is associated with the existence of an "urgent condition". The SIGIO signal is used with "interrupt driven i/o" (not presently implemented). SIGURG is currently supplied a process when out of band data is present at a socket. If multiple sockets have out of band data awaiting delivery, a select call may be used to determine those sockets with such data.

An old signal which is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any children processes have changed state. Normally servers use the signal to "reap" child processes after exiting. For example, the remote login server loop shown in Figure 2 may be augmented as follows:

```
int reaper();
 ...
sigset(SIGCHLD, reaper);
listen(f, 10);
for (;;) {
      int g, len = sizeof (from);

      g = accept(f, &from, &len, 0);
      if (g < 0) {
            if (errno != EINTR)
                  perror("rlogind: accept");
            continue;
      }
      ...
}
 ...
#include <wait.h>
reaper()
{
      union wait status;

      while (wait3(&status, WNOHANG, 0) > 0)
            ;
}
```

If the parent server process fails to reap its children, a large number of “zombie” processes may be created.