CONVEX Guide to Software Development

Document No. 710-001930-200

Version 3.0

November 2, 1987

CONVEX Computer Corporation

Revision/Update Information for CONVEX Guide to Software Development

Rev. No.	Date	Description
1.0	February 1985	First release.
2.0	July 1985	In addition to a few minor editorial changes, the following changes were made: Page 2-1 Added sentence 3 lines up from bottom. Page 2-3 Changed sentence 2 lines from top. Page 2-4 Added copy, last 2 lines of 2nd paragraph. Page 2-5 Changed 3rd example from bottom. Page 3-5 Changed "square" to "SQUARE" in 3rd example. Page 3-14 Changed last example on the page. Page 3-15 Added to 1st sentence, 4th paragraph from bottom.
3.0	October 1987	In addition to editorial changes, the following changes were made: Page 3-3 Changed sentence 2, of 1st paragraph. Page 3-3 Deleted paragraph at bottom of page. Page 3-4 Deleted list at top of page. Page 4-3 Changed sentence 1 of 3rd paragraph. Page 4-1 Added last sentence in 2nd paragraph. Page 5-1 Deleted sentence 2 of 3rd paragraph. Page 6-3 Deleted last section. Added Appendix A

Table of Contents

Appendices

List of Figures

Preface

Purpose

This document brings together an outline of some available tools and procedures for software development using the Berkeley UNIX system. The purpose is to provide enough information to familiarize a programmer quickly with the UNIX software development environment, and to allow programmers to make programs and systems that are

- Correct
- Debuggable
- Maintainable
- Usable
- Timely
- Predictable

This manual augments Kernighan and Pike's book, *The UNIX Programming Environment*, and provides cookbook examples of techniques used for system development. It concentrates on the Berkeley 4.2 implementation of UNIX and uses the Berkeley 4.2 C shell as the fundamental building block.

Intended Audience

System developers using the UNIX system at many levels can gain insight from this document. Most software examples are aimed at C programmers; Pascal and FORTRAN programmers can gain by using analogous constructs in their languages.

Document Structure

This document attempts to describe an environment in which system developers can easily understand and contribute to each other's software. In addition to standard environments, the manual describes some standard and not-so-standard but still useful techniques for programming. A chapter-by-chapter breakdown follows:

- Chapter 1 introduces the UNIX philosophy and defines software development.
- Chapter 2 describes the software development environments, including the shell, the file system, and the *make* program.
- Chapter 3 explains how to write and debug UNIX programs.
- Chapter 4 discusses three ways to decrease program execution time: profiling, optimizing, and recoding in assembly language.
- Chapter 5 describes various advanced techniques for solving software problems.
- Chapter 6 discusses the communications and revision control tools provided by UNIX.

- Chapter 7 describes various tools UNIX offers in aiding documentation production.
- Chapter 8 presents a brief conclusion.
- Appendix A describes the *contact* utility.

An index is included at the back of this guide.

Notational Conventions

The following conventions are used in this document:

- **Boldface** type indicates user-entered information.
- *Italics* designate commands, program names, switches, and variables.
- A horizontal ellipsis ... shows repetition of the preceding item(s).
- Brackets [ ] surrounding a set of characters indicate an option; i.e., you may choose any one of the characters.
- Braces { } are used to enclose a list of comma-separated items that can be expanded without regard to filenames.
- References to the *CONVEX UNIX Programmer's Manual* appear in the form *csd*(1), where the name of the manual page is followed by its section number enclosed in parentheses.

Associated Documentation

Throughout this manual you are referred to various documents that CONVEX provides:

- *CONVEX UNIX Tutorial Papers*
- *CONVEX Consultant User's Guide*
- *The C Programming Language*, Kernighan and Ritchie
- *CONVEX Adb Debugger User's Guide*

1

UNIX Philosophy

Many speak of the "UNIX philosophy," but rarely is it defined. This document concentrates on the development and use of simple, concise tools and the coordination of their output to minimize programmer effort while maximizing productivity.

The UNIX System

The UNIX system is more than just the kernel which schedules users and mitigates resource contention; it is the union of the operating system itself and the tools that it supports. Years of refinement by experts have brought it to the forefront of developmental technology. It is truly an expert friendly system. This document tries to aid the novice in attaining some facility to enjoy the expert friendliness.

The UNIX system's utilities are modular and useful both standing alone and when combined with each other. The UNIX mechanisms of pipes and file redirection are extremely useful. They allow the tools to be combined to form powerful conjuncts which can quickly solve typical problems encountered in the development environment. These tools let programmers think about programming while the machine does mechanical operations.

Plagiarism and careful augmentation are hallmarks of the UNIX philosophy. Simple, easy-to-use programs that do what they *should* comprise the basic UNIX building blocks.

Software Development

What is software development? It is the process embodying program conception, design, implementation, testing, debugging, and documentation. This document addresses all but the conception and design of software. It describes tools and techniques such as:

- A standard file and programming environment
- The *make* program
- Source program presentation
- Compiler invocation
- Lint
- Argument and environment processing
- Debugging
- File manipulation
- Capability files
- Screen manipulation techniques
- Profiling
- Optimizing
- Documentation hints, plagiarism as a tool for efficiency
- Configuration programs
- Shell programming
- Communications techniques
- Large project management

2

Software Development Environments

A standard environment for software developers saves effort and thinking time when they are developing a system. The knowledge that commands will "figure out what they're supposed to do and do it" frees the programmer's mind for considering more important items. The shell, the file system, and the *make* program can combine synergistically to yield a powerful software development environment. In this milieu, programs assume correct defaults for actions typically taken during a given project's lifetime; programmers can go to standard directories and find the object of their search; and typing *make* will most likely create "correct" results. A standard environment allows programmers to begin new projects and switch among current ones, confident that things are "as they should be." Their work can proceed without time lost searching for files or creating new techniques when standard ones will do.

Shell Programming

Some claim that the most overlooked, high-level programming language is the command interpreter (usually called "the shell"). On BSD UNIX systems, there are two shells: the Bourne shell (named *sh*, the prompt is usually $) and the C shell (named *csh*, usually with the % prompt). Many articles and tutorials document the Bourne shell; this section describes *csh*. Both shells contain similar control structure functionality; it is usually possible to achieve similar results on another shell with only minor reworking of the shell's input.

The paper "An Introduction to the C Shell" by William Joy gives an excellent tutorial on the basics of *csh* (see the *CONVEX UNIX Tutorial Papers*). Kernighan and Pike also provide several good examples. Included here are a few skeletal scripts that are useful for building solutions. Be sure to include the '#' character as the first line of a *csh* script. The single-most common mistake in writing *csh* scripts is the omission of the # symbol as the first character of the file. When command executors encounter a shell script *without* the #, they invoke the Bourne shell (*sh*) to process the script, often with unacceptable results. The latest acceptable style dictates that *csh* scripts begin like this:

```
#!/bin/csh
```

or, if the *.cshrc* file need not be read, like this:

```
#!/bin/csh -f
```

Later instances of # denote the start of a comment, which continues to the end of that line.

Probably the shell script used most often is the "do this for each of a list of files" script. Here is one incarnation:

```
#!/bin/csh -f
foreach i ($argv)
        if ( -f $i ) then
                echo processing $i\:
#               do the processing
        else
                echo $i does not exist
                continue
        endif
end
```

The 'foreach' loop sets *i* successively to the values of the arguments to the shell script as the loop is iterated. The *if* statement chooses one of the alternatives based on whether the current argument ($*i*) is a plain file or not. The blanks surrounding the parentheses in the *if* statement are important syntactic delimiters—do not omit them.

Here is a more complicated script which demonstrates the rest of the control structures. It scans its arguments for various flags before gleaning the numeric month and day from the output of the *date* command:

```
#
#
#
# have we set the date via the -D switch?
if ( $1 == "-D" ) then
  set date = $2
  egrep "ate-" /usr/spool/uucp/SYSLOG | awk -f /usr/lib/uucp/uusum.awk
else
  set people = $argv
  if ( $people == "" ) then
    set people = "root"
  endif
  set date = ('date')
# sample date command output:    Sun Apr  1 15:14:19 CST 1984
# third field is the day of the month:
  set day = $date[3]
# second field is the alphabetic month:
  switch ( "$date[2]" )
        case 'Jan':
                        set mon = 1; breaksw;
        case 'Feb':
                        set mon = 2; breaksw;
        [ ... obvious text omitted ... ]
        case 'Dec':
                        set mon = 12; breaksw;
  endsw
# search the uucp logfile for lines with the appropriate date:
  egrep "\($mon/$day-" /usr/spool/uucp/SYSLOG |\
        awk -f /usr/lib/uucp/uusum.awk |\
      /usr/ucb/mail -s "Daily UUCP Summary for $date" $people
endif
```

This script uses an *awk* program to examine portions of the UUCP logfile based on the date desired. It includes both an *if* and a *switch* statement as prototypes.

To avoid creating a complex shell script, use the *apply* command. It causes the shell script or command that is its first argument to be invoked successively with each of the succeeding arguments. Here is a single-line command which invokes the *indent* command on each of three files:

apply indent a.c prog.c subr.c

For both shells, typing filenames is simplified. It provides a form of regular expressions that expand to match names occurring in the file system. These expressions have slightly different rules from those found in the editors or in the *grep* family. The shell file-expansion mechanism (*glob*) allows * to match any number of characters (i.e., 0 or more) and ? to match any single character. The following session transcript illustrates the use of * and ?:

```
% ls
1               4               7               argstds         makefile.so
1.out           4.out           7.out           argstds.more    outline
2               5               8               handbook        outlinemac
2.out           5.out           8.out           macro           sp.6
3               6               README          makefile
3.out           6.out           a               makefile.num

% echo 1*
1 1.out

% echo *.out
1.out 2.out 3.out 4.out 5.out 6.out 7.out 8.out

% echo out*
outline outlinemac

% echo *out*
1.out 2.out 3.out 4.out 5.out 6.out 7.out 8.out outline outlinemac

% echo ?
1 2 3 4 5 6 7 8 a

% echo ?.out
1.out 2.out 3.out 4.out 5.out 6.out 7.out 8.out
```

Brackets surround a set of characters, any one of which may match for a filename to be qualified. Here's a sample from the directory shown above:

```
% echo [1234].out
1.out 2.out 3.out 4.out
```

Braces surround a list of comma-separated items (of any length) that are expanded without regard to file names, e.g.:

```
% echo Ralph{1,2,3}
Ralph1 Ralph2 Ralph3
% echo {a,b,c}{d,e,f}
ad ae af bd be bf cd ce cf
```

Debug C shell scripts with the *-x* and *-v* options. The *-v* option causes commands to be echoed immediately after history substitution; the *-x* option causes commands to be echoed immediately before execution.

Shell Environmental Aids

The shell does its best to make things easy for programmers. It reads the .cshrc file in the user's home directory as it is started. If it is a login shell, after reading .cshrc, it also reads and interprets the .login file from the user's home directory. These files can contain any shell commands and are typically used to establish aliases and shell environmental variables.

Aliases

Aliases can save lots of typing for commonly used commands. They are easy to enter (into .cshrc) and have a simple format. The following line provides the *C* command which invokes a line editor on a file of mail-routing connections:

alias C ed /usr/src/usr.local/makeuumap/connects

The existence of a single-letter command eases the fear of typing the command incorrectly and can encourage its use. It also can provide disastrous results for typographical errors. Display all aliases by typing

alias

alone on a line.

Directory Traversal

Two aliases aid users in moving among directories and complement the *pushd* and *popd* commands:

```
alias cd   'set old=$cwd; chdir \!*'
alias back 'set bk=$old; set old=$cwd; cd $bk; unset bk; dirs'
```

These aliases appear in the .cshrc file since they are useful in all shell instances. They augment the *cd* (change directory) command's function by remembering the previous working directory. The new *back* command returns to that previous directory. Unlike *pushd* and *popd*, these aliases remember only one level of directory travel.

The directory stack manipulation commands, *pushd* and *popd*, combine the function of the *cd* command with a stack-like memory. Each time you invoke *pushd* with a directory name, the shell changes the working directory to that name and pushes it onto the directory stack. The *dirs* command prints the current directory stack; the current working directory is listed first (and also stored in the shell variable *cwd*). The *popd* command discards the current directory from the top of the stack and does a *cd* to the directory that forms the new top-of-stack.

Additionally, *pushd* +n rotates the directory stack to bring the nth entry to the top and *cd*'s there. The directories are referenced as though they started with zero. Consider this example:

```
% dirs
/mnt /usr /tmp / /docs/sofdev
% pushd +2
/tmp / /docs/sofdev /mnt /usr
% popd
/ /docs/sofdev /mnt /usr
% pushd +3
/usr / /docs/sofdev /mnt
```

You can also specify *pushd* with no arguments. The shell then exchanges the top two stack members (while *cd*'ing to the new top-of-stack). Specifying *popd* +n discards the nth argument from the stack.

Command Directories

The shell searches not only the home directory but also others for the target of a *cd* request if the shell variable *cdpath* is set. Adding the line

set cdpath = (. /sys/sys ~ ~/docs)

to *.cshrc* causes the shell to search the current directory, the UNIX system directories, the home directory, and finally the docs directory under the home directory for a directory name matching the *cd* specification. Only if no match occurs throughout the search does the *cd* command fail. The *cdpath* command allows you to envision a set of directories as being "globally available" as targets of a *cd* command.

The shell sequentially searches several directories for command names. Setting the path shell variable (which is automatically inherited by subshells) in *.login* specifies the sequence of directories to search for a command:

**set path = (. ~/.commands /usr/local/bin /usr/ucb **
/bin /usr/bin /etc /usr/parsec/X/bin /usr/68k/X/bin)

Note both the personal command directory (~/.commands) near the front and the special project directories at the end of the search list. It is a matter of preference whether the current working directory (.) goes at the beginning or end of the path.

The *which* command tells which executable file the shell chooses for a particular command. For example,

which ls

evokes the response

```
/usr/ucb/ls
```

Explicit path names (e.g., /bin/ls or ./ls) override the search order. Be sure to enter *rehash* after adding a new command in one of the command directories (unless you log out). Otherwise, the shell will not find the new command, since the shell really only looks for commands once, when it is started (and then remembers where they are).

Shell and Environment Variables

Some programs examine certain variables from the environment to customize their performance for users (e.g., *mail* strives to give you your favorite editor for editing outgoing mail). Set these variables in *.login*. Probably the most important is the EDITOR variable:

setenv EDITOR /usr/ucb/vi

Other programs require their own specific environment variables. Read each program's documentation to see if it can be made friendlier for some user's tastes.

Shell variables and environment variables can be handy. Using

set notify

in *.cshrc* causes the shell to print results of background jobs (e.g., "job completed" or "job waiting" for tty input) as soon as the results are available, instead of waiting for the user to encounter a shell prompt. The mail variable (also in *.cshrc*)

set mail = "/usr/spool/mail/smith"

tells the shell to be on the lookout for new mail. As new mail arrives (i.e., the file update time is changed), the shell prints a notification (though for efficiency the file is checked only once every few minutes). Specify the checking interval by a numerical argument before the filename:

set mail = (30 /usr/spool/mail/smith)

to check mail whenever a shell prompt is to print later than 30 seconds since the previous check.

Some users despair of hitting control-D one too many times and finding themselves logged out. Set the shell variable *ignoreeof* to tell the shell to demand that you type *logout.*

The redirection of output to already extant files (or appending to nonexistent files) can be eliminated by setting the shell's *noclobber* variable. Override the specification with the >! and >>! redirection operators (the ! must be escaped by a \ -- see "History Mechanism").

Programs that execute for longer than the shell's *time* variable's value in seconds automatically cause a summary of their execution time to be printed. If the *time* variable's value is to be 10 seconds:

set time = 10

then summary lines for programs that run longer than 10 seconds are shown. Here is a sample summary line:

138.7u 57.4s 2:31:25 2% 74+117k 807+659io 728pf+14w

which indicates:

- 138.7 user seconds (time spent executing programs in user space)
- 57.4 system seconds (time spent servicing system call requests)
- 2 hours, 31 minutes, and 25 seconds total elapsed time of program execution
- 2% of the CPU time consumed during the program's lifetime
- 74 kilobytes average use of shared memory
- 117 kilobytes average use of unshared memory (data space)
- 807 input operations
- 659 output operations
- 728 pages faulted into memory
- 14 pages written out of memory

The *prompt* shell variable can be set to change the normal shell prompt from % to any other fixed string. You can also specify that the command number (see "History Mechanism" below) be included in the prompt. It is substituted for the ! character:

set prompt = "(\!)% "

causes a parenthesized command number (which eases later reference to commands) to precede the normal % prompt. Note the space after the % and the escaped !. The escape is necessary to subvert the history mechanism (see below).

History Mechanism

In its continuing effort to be helpful, the shell's history mechanism remembers commands typed in the past. The shell variable called *history* should contain some value representing the number of commands to retain during the current shell's session. The shell variable *savehist* specifies the number of commands to retain across login sessions. While it is tempting to assign a large value to *savehist,* resisting that temptation will favor increased performance for subshells since the saved history file is read for every subshell (including subshells that just read shell scripts). A typical *.cshrc* file might contain:

set history = 200
set savehist = 25
alias h history 23

The *h* alias allows a single-letter command to display exactly one screen of history. This display aids the user in repeating commands since the shell makes it so easy to reissue them. Just type *!* followed by either the number of the command to repeat or the first letters of the command to repeat (as found by the shell by searching backwards through the history list). Consider a short development cycle that repeats the following commands:

```
% vi test.c
% cc -o test test.c
% test -f alpha -e4
```

It is sufficient to reinvoke these statements as

```
% !v
% !c
% !t
```

thus saving typing time and eliminating (or propagating!) typographical errors. The ! history reference may occur anywhere within the command string. Escape the exclamation point (\!) when you wish to pass the exclamation point itself to the shell (e.g., for new shell prompts, I/O redirection override, or UUCP addressing).

The shell allows you to modify a command specified as above by entering a colon and substitute string after the specification. If you had just compiled *ace.c* with:

```
% cc -O ace.c libfile.o libfile2.o libfile3.o
```

and wished to compile *deuce.c* with the same library files, then:

```
% !cc:s/ace/deuce/
```

would do the trick (as would *!cc:s/a/deu/*). This mechanism aids correction of spelling errors in long commands (though often one must correct the command using more than one line since it can only do a single substitution). Appending *:p* to a substitute command causes the resulting modified command to be printed and entered into the history list but not executed:

```
% !cc:s/ace/deuce/:p
```

It then allows the modified command to be modified itself. Substitute *gs* for the *s* directive if the change is to be made globally throughout the command instead of just once.

The command

```
% ^alhpa^alpha
```

abbreviates the often needed command

```
% !!:s/alhpa/alpha/
```

thus simplifying immediate spelling correction.

The shell also provides handy meta-file names: *!** denotes all command arguments on the previous line, *!1* denotes the first argument, *!#* denotes the #th argument, and *!$* denotes the last command argument on the previous line. This allows the typing of

```
% echo some*files*
% rm !*
```

as a way to be sure which files matched by the wildcards will be deleted. Consult *csh*(1)'s section "History Substitutions" for fancier substitution and argument handling.

Redirecting Current Shell Input

Commands such as *cd*, *alias*, and those involving history substitution work as built-in shell commands. For each command, a new process is NOT forked; rather, it is interpreted by the shell. This is a clear implementation requirement for commands such as *cd* which serve only to modify internal shell variables. Normally, shell files are interpreted by a subsidiary shell forked for the user's benefit by the current command interpreting shell. If these shell files contain such commands as *cd*, which the user wishes to have influence on the current shell, the change is lost as the subsidiary shell terminates.

To cause the current shell to read its input from a file instead of the terminal (and hence to interpret commands from a file and retain their influence on the shell), use the *source* directive. As an example, the commands

```
% source ~/.cshrc
% source ~/.login
```

perform the same function as the shell automatically does at login time.

Initializing the Terminal During Login

The *.login* file can also properly initialize your terminal and set other miscellaneous functions. Typically, customized calls to *stty* and *tset* appear. In this example, *stty* informs the system that the terminal can erase characters with backspace. The *tset* command does any necessary terminal mode initializations (as determined by the terminal type and the *termcap* database):

```
stty crt
tset
```

The *biff* command notifies the system that the user wishes instant notification when new mail is received. The system then displays the first few lines of the message (unlike the quieter "You have new mail." of the shell's *mail* variable) upon arrival of new mail, thus interrupting whatever typing or printing is in progress. To enable *biff*, specify the following in *.login*:

```
biff y
```

To learn if new system-wide messages have arrived since the previous login, enter the following in *.login*:

```
msgs -q
```

The shell and other programs use environment variables and initialization files to aid system use. Emulation of their examples will aid production of good software in the future.

Job Control

The C shell has a powerful mechanism for job control: the ability to send a job to "background." Normally, it is easy enough to use the & to send a job to background; but what if you want to move it back to foreground (e.g., for teletype input)? The C shell provides the ^Z (control-Z) feature. Typing ^Z suspends the current job (it is neither executing nor terminated) and prints the message "Stopped". You can then send the job to background with the *bg* directive or bring the job back to foreground via *fg*. The *jobs* directive gives the status of all jobs currently suspended ("Stopped") or "Running" in background. Invoking either *fg* or *bg* with *%n* (n represents the reported job number) sends a job into foreground or background, respectively. To stop a background job (i.e., suspend it), bring it to foreground then type ^Z.

The ^Z command is particularly useful when perusing several files with an editor. Type ^Z when ready to perform a peripheral task; *fg* to bring the editor back later.

Note that some programs do not work so well with ^Z because they use raw teletype input/output.

Note also that %n is shorthand for *fg* %n and %n& is shorthand for *bg* %n.

File System Environment

Finding needed files quickly within a large project directory can decrease software development time and reduce frustrations. This section describes a standard way of naming source, object, and documentation files so that project users can quickly find any needed information.

Standard Extensions

The UNIX file system allows files to have virtually any name. Some utilities, however, respect those letters of a filename after the last period. By naming files in a standard way, developers can know with just a glance what kinds of files are in a directory. Here are some of the standard extensions:

.c	— C source code
.h	— #define's for source code
.f	— CONVEX FORTRAN source code
.o	— Native object code
.s	— Native assembler code
.ar	— A file containing files combined by *ar*(1)
.a	— A file containing files combined by *ar*(1)
.tar	— A file containing files combined by *tar*(1).
.awk	— Input to the *awk* processor
.l	— Lex source code
.l	— Libraries for the *bc* processor
.y	— Yacc source code
,v	— RCS archive files (usually preceded by normal extension)

Some developers use other extensions. Their use is usually clear (e.g., extension *.nr* for an nroff source file).

Filenames

The *ls* command sorts its filename output such that filenames beginning with digits appear first, then those beginning with uppercase letters, then those starting with lowercase letters. This property is occasionally used to highlight some files by bringing them to the front of the listing. The README file is a typical example of this practice.

Standard Directories

Each project has its project name as its highest-level directory. All files relevant to the project are contained somewhere beneath that directory. Project directories often include the following directories and files as its constituents (unneeded directories are, of course, omitted):

- README: a quick description of the files in the directory
- doc/: descriptions, tutorials, documents relating to the project
- man/: manual pages for the project (with appropriate suffixes)
- src/: source and object programs (sometimes in subdirectories)

- h/: include files (sometimes this is called "include")
- lib/: project subroutine libraries and miscellaneous data files
- test/: verification suites and test programs and data
- makefile: a master makefile (see chapter on makefiles)

Each of the subdirectories should also contain a README file with a brief description of the directory's contents and how its makefile works.

The system program *whereis* has a mechanism which efficiently searches many directories for a specified file with any of a number of extensions. Large projects may find utility in producing a modified copy of this program for large directory structures.

Ownership/Protection

In many projects, several people work on the project's various files. Overprotecting the files makes those people's work difficult. It is recommended that users in these circumstances set their umask to 002 in *.login*

umask 002

to inhibit those users not in the project's group from modifying the project's files. The first two 0's give read, write, and execute access to the owner and those members of the same group as the file. Ensuring that those working on a project are in the appropriate group minimizes protection and ownership problems.

Make

The *make* program is among the most powerful tools for software development on the UNIX system. Essentially, *make* has as its input both a makefile and the modification times of all files referenced by the makefile. This makefile has a set of rules and instructions which tells *make* how to achieve its goal (usually by recompiling some source programs and linking them). Because the UNIX linker is very fast (at least compared to compiling hundred-line files), it is convenient to use it to link a large number of modules. Until the number of modules causes *make*'s overhead to become unacceptable, it is usually efficient to keep module size small.

make uses a set of rules to perform the smallest set of transformations to create a set of output files from a set of input files (e.g., a set of linked object modules from a set of source programs). Because it has a dependency list as its input, *make* can usually avoid unneeded recompilations.

make has internal variables which increase the flexibility and maintainability of makefiles. The current trend in many shops is to isolate all customizable compile-time parameters (e.g., maximum number of system users) into the directory's makefile so that it alone is the focus of parameter maintenance.

In general, makefiles will contain sets of variables (often to specify parameters to programs or define items to be used repeatedly throughout the makefile), specifications, and dependencies (which items must be updated to create a specification and how to do the actual update). The *make* program has a set of implicit transformations which tell it how to, for example, compile a file with *.c* extension into one with *.o* extension. It is also possible to provide explicit general rules for *make* to use when setting up a system. To remove the normal internal rules and proceed without them, use:

make -r ...

You can specify new "built-in" rules by specifying a *SUFFIXES* statement and a set of rules for converting from one suffix to another. Here's an example of a *make*'s internal rule for compiling C programs:

```
SUFFIXES = .o .c

        $(CC) $(CFLAGS) -c $*.c
```

In the context of the second and subsequent lines,

- $* represents the root of the name to be transformed
- $< represents the name with the initial extension (suffix)
- $@ represents the name with the target extension
- $? represents all the files specified as dependencies

Unfortunately, these expansions do not work on the dependency line.

Modern makefiles are expected not only to be able to compile (or recompile) source libraries but also to:

- Effect installation of the package
- Use *lint* for strict checking of appropriate C sources
- Remove files normally mechanically created
- Ascertain header-file dependencies of source files
- Make a listing of source programs and headers
- Execute a testing and/or verification suite

Following is a prototype makefile that performs these tasks. Read it for form rather than detail:

```
1   #       Thanks to Ray Essick at University of Illinois
2
3   #
4   #  User Changeable Parameters:
5   #
6
7   BIN     = /usr/local/bin
8   MSTDIR      = /usr/spool/notes
9   NOTESUID = 10
10  NOTESGRP = uucp
11  CC      = cc
12  CFLAGS      = -DMSTDIR=\"$(MSTDIR)\" -DNOTESUID=$(NOTESUID) \
13              -DBIN=\"$(BIN)\" -I../h
14  LFLAGS      = $(CFLAGS)
15
16  # end of User Changeable Parameters:
17
18  UTILITY = $(MSTDIR)/.utilities
19
20  HFILES      = parms.h structs.h
21  HLPFILES    = access.help dir.help
22  CFILES      = access.c adnote.c adresp.c main.c
23  OFILES      = access.o adnote.o adresp.o main.o
24
25  CMDS        = main
26
27  bin:    main
28          @echo notesfile binaries up-to-date
29
30  clean:
31          rm -f *.o $(CMDS)
32
33  sizes:  $(CMDS)
```

```
34          size $(CMDS)
35
36  help:
37          @echo type make install to do everything
38
39  spool:
40          -mkdir $(UTILITY)
41          -chown $(NOTES) $(UTILITY)
42          -chgrp $(NOTESGRP) $(UTILITY)
43          chmod 755 $(UTILITY)
44
45  install: $(HLPFILES)
46          cp $(HLPFILES) $(UTILITY)
47          (cd $(UTILITY); chmod 644 $(HLPFILES))
48          @echo ------ doc installation done
49
50  #       User programs
51
52  main:   OFILES
53          @echo loading main
54          @cc $(CFLAGS)  -o main $(OFILES)
55
56  print:
57          pr -f $(CFILES) $(HFILES) | lpr -b "Notesfile Code"
58
59  tags:   $(CFILES)
60          ctags $(CFILES)
61
62  lint:   $(CFILES)
63          lint $(LINTFLAGS) $(CFILES)
64
65  test:
66          @echo Sorry, I do not know how to test anything.
67
68  depend:
69          grep '^#include' ${CFILES} | grep -v '<' | \
70              sed 's/:[^"]*"\([^"]*\)".*/:  \1/' | \
71              sed 's/\.c/.o/' | sed 's,^[^     ]*/,,' | \
72          awk ' { if ($$1 != prev) { print rec; rec = $$0; prev = $$1; } \
73          else { if (length(rec $$2) > 78) { print rec; rec = $$0; } \
74               else rec = rec " " $$2 } } \
75              END { print rec } ' > makedep
76          echo '$$r makedep' >>eddep
77          echo '/^# DO NOT DELETE THIS LINE/+1,$$d' >eddep
78          echo '$$r makedep' >>eddep
79          echo 'w' >>eddep
80          cp makefile makefile.bak
81          ed - makefile < eddep
82          rm eddep makedep
83
84
85  # DO NOT DELETE THIS LINE -- make depend uses it
86
87  main.o: structs.h parms.h
```

Lines 7 through 14 of the makefile contain the user configurable parameters. This makefile prefers to pass them directly to various C programs using the *-D* command line option. Alternate techniques include generation of the makefile from some more abstract configuration file or a scheme like this:

```
start:  makefile
        rm -f parms.h
        echo #define PARMA $(PARMA) >> parms.h
        echo #define PARMB $(PARMB) >> parms.h
```

where file *parms.h* is included in those routines needing the changeable parameters. Any of these schemes focuses parametric changes in a single file. It may be possible to echo the new parameters into a new file which can be compared to the old file to check for changes. If none exist, the makefile can simply touch certain files to make them appear updated without recompiling large numbers of programs dependent on the parameter file.

It is interesting to note that define statements for input to the C preprocessor (such as user identification numbers) are included as makefile parameters. Additionally, such parameters as installation directories and compile-time options (including which compiler to use) are also included in the makefile. Note that the *CFLAGS* variable includes the -I../h specification which centralizes to a single place the rule for searching of header files.

The file classification section (lines 18-23) groups similar files into slightly more general abstractions. Often this is the only place specific filenames are mentioned. The philosophy of "things only show up once" aids in long-term maintenance.

The first item that causes work to be done is *bin* (lines 27-28). Because it is first, *bin* is the default specification when a user types *make* with no parameters. In this example, *bin* does all compilations and brings the system up to the point where installation is possible. The *clean* entry (lines 30-31) does just the opposite: it returns the directory to its virgin status.

Invoking *make sizes* (see lines 33-34) will report the lengths of each command. Similarly, *make print* (lines 56-57) prints the files; *make tags* (lines 59-60) creates a tag file for the *vi* editor. The *help* (lines 36-37) specification is trivial and, in this case, useless. This particular makefile provides the ability to create a set of spool directories via the *spool* specification (lines 39-43). Installation proceeds with *make install* (lines 45-48).

Finally, this makefile includes the powerful *depend* (lines 68-82) specification. It is not necessary to understand the constituents of these lines, only the result. This command set searches each of the source files for *include* statements. It then massages those statements and appends them to a copy of the makefile, thus automating the process of determining dependencies within include files. Dependencies on nested include files are NOT detected using this scheme. Note also that this *make depend* specification will not work if only one file is specified for *CFILES*, since *grep* will not put that file's name in front of each output line.

This canonical makefile shows only the minimum number of files and specifications to give a flavor of a project's makefile. You may extend or change each of the file listings and parameters. Use this example as a base upon which to build.

Feel free to nest makefiles in directories below a master makefile and "recursively" invoke *make*. Also feel free to subvert *make*: its idea of how to interface with *RCS* is unfortunate.

It is possible to specify an internal variable to *make* on the following command line; this facility is useful in the experimental development environment:

```
make BIN=/usr/local/bin
```

Debugging makefiles can be a trial when they become complex. Specifying the *-n* switch to *make* causes it to report what operations would be performed—but without performing the operations.

Specifying the *-t* switch causes *make* to update modification times on files that would be modified if *make* were invoked with the specified object. No commands are invoked (e.g., compiles) when the file modification times are updated. The *-p* switch causes *make* to print out the definitions as it understands them. Specify *-ts* to request *make* to touch all its files silently (similarly to *-t* but with no notification).

When at rope's end in debugging makefiles, use the *-d* switch to obtain verbose debugging information about the rules specified, the search lists, and the file update times.

3

Writing and Debugging UNIX Programs

This chapter presents a potpourri of techniques which extend the ideas of uniform environments to uniform programming techniques. Source code presentation, argument processing, and debugging are among the suggestions for standardizing certain internal aspects of software. The chapter also includes several miscellaneous hints for UNIX/C programming techniques in addition to some ideas for debugging UNIX programs. Most concepts in this chapter deal with C programs; some are applicable to Pascal and FORTRAN.

Source Programs

Source Code

The ability to see a program's structure quickly can aid in spotting errors and understanding new code. Both horizontal white space (around operators and some variables) and vertical white space (between blocks and sometimes within loop structures) aid in presenting a program. The *indent* program reformats C source code to conform to a set of coding standards. This program has several options to customize its behavior (e.g., varied line lengths and brace placement). It is advantageous for a project or even an entire shop to adopt a uniform indentation policy. This allows programmers to read each other's code without confusion concerning indentation "style."

Headers

Because modules can often be reused among different projects and programmers often must decipher what a module does, each module should contain some kind of a header. Although many times these headers contain only a one-sentence description of the routine, good headers contain enough information to understand the routine in short order. Figure 3-1 illustrates a prototype header for a sort subroutine.

The header's categories should be self-explanatory. The header in Figure 3-1 allows the maintainer to know:

- What the routine does
- What its parameters are
- What it returns
- What external dependencies it has
- Exactly whom to contact about problems

It is easy to keep a header template in an easily accessible directory to be inserted with a simple editor command.

Figure 3–1: Sample Header for Sort Subroutine

```
static char *Copyrightetc = ''Any copyright or whatever data'';
/*=======================================
 *
 * NAME:       sort.c
 *
 * PURPOSE:    sorts an array of integers into ascending order
 *
 * ALGORITHM: Bubble Sort (slow but sufficient for this application)
 *
 * PARAMETERS:       sort(a,n)
 *             int a[]; -- array of integers to be sorted
 *             n -- number of integers in a
 *
 * RETURNS:    Nothing
 *
 * GLOBALS:    None
 *
 * CALLS:      None
 *
 * CAUTIONS:  Lists can be no longer than 100 elements
 *
 * BUGS:       None known
 *
 * HISTORY:
 *      initial coding -- R. Kolstad -- 2/19/84
 *
 *=======================================*/
```

Comments

Most shops encourage the use of comments. While discouraging those comments such as:

```
i = 1;          /* set i to 1 */
```

others can be very useful:

```
for (p = head; *p; p = p -> next)  /* traverse the linked list */
{
   /* process item p */
   ...
}
```

Aids and Utilities

Many people have written programs which aid in programming with the C language. One of the more useful ones for understanding declarations is *cdecl*. This program converts C declarations to English and vice-versa. Here is a sample of running *cdecl* (user input is in boldface):

```
% cdecl

explain int **(*a)(b, c)
declare a as pointer to function (b,c) returning pointer to pointer to int

declare a as function (b, c) returning pointer to function returning int
int (*a(b,c))()
```

cdecl can eliminate errors due to mismatching of parentheses and asterisks in tricky pointer declarations. See *cdecl*(1) for more information.

The *cchk* program (written by Steve Draper at UCSD from a base provided by Tom Anderson of Fluke and Jeffrey Mogul of Stanford; distributed via the newsgroup *net.sources*) examines source code for typographical errors commonly encountered in C. These include:

- Correct matching of all kinds of brackets
- Indentation, semicolon checking
- "Dangling else" errors
- Nested comments
- Assignment instead of equality within *if* statements
- Empty control statements due to inadvertent semicolon placement
- Premature end of statement due to missing braces

The *cchk* program quickly points out errors typical of the neophyte and hence is very useful for beginners and experts alike.

The *cparen* program aids in interpreting C precedence. Written by Brad Needham of Tektronix and distributed via *net.sources*, it prints a fully parenthesized version of a valid C expression; that is, it shows the operators' precedence with parentheses. Here is an example (user input shown in boldface):

```
% cparen
a+b*c>>5%6|4
((a+(b*c))>>(5%6))|4
%
```

Cross-Reference Listings

Unlike many other compilers, the UNIX C compiler produces no listings (source code or otherwise). To programmers used to cross-reference listings or those requiring cross-references for large projects, this loss can have an adverse impact. The *cref* program (written by Steve Zimmerman and published via the newsgroup *net.sources*) provides these listings. *cref* generates a complete cross-reference listing of a set of C programs.

The *calls*(1) program (written by M. Taylor of DCIEM in Toronto and modified by Alexis Kwan of Human Computing Resources at DCIEM) analyzes control flow of subroutine calls.

Editing Large Systems

The *vi* and *ex* editors can jump among many various source files within a large project by using the *ctags* program to generate a file named "tags". This file tells where each major C or FORTRAN program identifier is defined. This file is referenced by the editors when a *:ta tagname* command is issued. The editor searches the tags file for "tagname", and upon finding it, edits the file containing the definition and sets the cursor to the beginning of the definition. An alternate way to invoke the *:ta* function is to place the cursor at the beginning of a reference to the identifier and use the ^] character (control-right bracket).

Printing Sets of C programs

The *cpr*(1) program provides a useful way to print C programs on hardcopy devices. It separates modules appropriately and prints an index of where each module resides. See *cpr*(1) for details.

The Preprocessor

The preprocessor scans each program just before the source program enters the compiler. It processes statements to:

- Include other files
- Define (and expand) macros with possible arguments
- Conditionally accept source code

It is good programming style to use the *define* facility to give explicit names to all nonobvious constants or "magic numbers" (those with absolute value other than 0, 1, or possibly 2). Some shops suggest that such constant defines have names consisting of capital letters, e.g.:

```
#define     PI      3.1415926535
#define     BUFSIZE     2048
```

It is sometimes desirable to write code which is only used for some compilations. The *ifdef* facility allows programmers to create files which include some code only if the object of the *ifdef* exists, for example:

```
#ifdef vax
#  define PAGESIZE 512
#else
#  define PAGESIZE 1024
#endif
...
    pagesize = PAGESIZE;
```

The preprocessor automatically defines the parameter *vax* for all compilations on DEC's VAX computers; other constants are defined for other machines.

Parameterized macros can remove clutter and repetition from user programs while concentrating complicated arithmetic operations and understanding in a single place. Here is a macro define to calculate the logarithm base 2 (to the nearest integer) of a small integer:

```
#define LOG2(x)  ((x)<=2?1: (x)<=4?2: (x)<=8?3: (x)<=16?4:      \
                  (x)<=32?5: (x)<=64?6: (x)<=128?7:          \
                  (x)<=256?8: (x)<=512?9: (x)<=1024?10:      \
                  (x)<=2048?11:12 )
```

Note the enclosure of the argument *x* by parentheses. This ensures that the arithmetic expression parsed by the compiler contains the expected precedence. Because the compiler evaluates

constant expressions at compile time, this macro is an efficient way to compute the number of bits needed to store a constant expression whose value is less than 4096.

Nested *include*'s create some problems. Dependency lists for the makefile are already error-prone. Nested *include* files aggravate this problem by masking changes to a file's meaning by modifying an altogether different file. One trick to clarify whether a particular nested *include* has shown up is to use the *ifndef* facility to bypass redefinition of a given token. Consider the example below:

```
#ifndef PAGESIZE
#include "vmparameters.h"
#endif
```

If *PAGESIZE* has already been seen, it is assumed that the file "vmparameters.h" has already been included; otherwise, the file is read and *PAGESIZE* becomes defined.

Beware of side effects when invoking defined functions. Consider:

```
#define SQUARE(x)  ((x)*(x))

    ...
    a = SQUARE(i++)
```

The square define expands to a wholly unexpected expression which gives (usually) an unacceptable result.

Invoking Compilers

Most compilers on UNIX recognize relevant extensions of their arguments. Invoking the C compiler having files with extensions of .c, .s, and .o causes appropriate compilations, assemblies, and links, respectively. Likewise, the FORTRAN compiler accepts *.f*, *.s*, and *.o* extensions. The standard UNIX FORTRAN compiler runs the preprocessor on files with extension of *.F* . This feature obviates the need to become an expert at the loader and program libraries until absolutely necessary.

Few UNIX compilers produce program listings; most report errors via line numbers. The *error* program combines these error reports back into source file text for easy program modification. Since error output appears on the "standard error" file descriptor, ampersands must be used for correct indirection. Here is an example of a small, erroneous C program:

```
integer i;
main ()
{
   i = 1;
   printf ("Hi there, i is %d\n", i)
}
```

Invoking the C compiler and *error* program yields output like this:

```
% cc a.c |& error
1 file contains errors "a.c" (4)

File "a.c" has 4 errors.
        4 of these errors can be inserted into the file.
You touched file(s): "a.c"
%
```

File a.c now contains both the program and the errors:

```
/*###1 [cc] warning: undeclared initializer name i%%%*/
/*###1 [cc] illegal initialization%%%*/
/*###1 [cc] warning: old-fashioned initialization: use =%%%*/
integer i;
main ()
{
   i = 1;
   printf ("Hi there, i is %d\n", i)
/*###6 [cc] syntax error%%%*/
}
```

These errors correctly reflect the invalid declarator **integer** and the missing semicolon after the *printf* statement.

Lint

The C compiler does no strong type checking. In fact, because modules are usually small and separately compiled, generally the compiler cannot check whether parameter types of the function invocations match those of the function's definitions. The loader does no such checking, either. The *lint* program (so called because it is so picky) provides a way to check for such consistencies. It reads an entire set of source programs and prints error and warning messages about possible inconsistencies in function invocations, type mismatches in expressions, and other kinds of subtle errors. It is usually a good idea to make programs "*lint*-free" before investing too much time debugging them. Some shops go so far as to issue an 11th commandment:

XI. Thou Shalt Use Lint

Avoid the temptation to save time in the short run; use *lint*.

There are several ways to inform *lint* that certain statements are written correctly (according to the programmer). These include the *ARGSUSED* and *NOTREACHED* directives. They are enclosed in comment markers and placed before the statement described. See *lint*(1) for more details.

Arguments and Environments

Many UNIX programs require slight customization for each use. For example, you can direct the *ls* command to list filenames alone or filenames together with protections, owners, and modification dates. Three ways to implement this parameterization are arguments, environment variables, and initialization files.

Arguments

Most UNIX commands have evolved to use a standard set of conventions for invocation. A.T.&T. Bell Laboratories has formalized and distributed these standards. They are:

- Command names must be between two and nine characters.
- Command names must include lowercase characters and digits only.
- Option names must be a single character in length.
- All options must be delimited by "-".

- Options with no arguments may be grouped behind one delimiter (e.g., *-lsx* specifies three options).
- The first option-argument following an option must be preceded by white space (i.e., if the *-f* options expects a filename, that filename must appear as a separate argument).
- Option arguments cannot be optional (i.e., if the *-f* argument expects a filename to follow it, that filename MUST be included).
- Groups of option-arguments following an option must be separated by commas or by white space and quoted (i.e., if an option requires more than a single word as an argument, the option list must appear to the shell as a single word. Both a,b,c,d and the quoted "a b c d" are parsed as a single word by the *shell*. Inserting white space after the commas in the a,b,c,d example would yield four arguments instead of one).
- All options precede operands on the command line.
- '--' may be used to delimit the end of the options. Otherwise, if one wished to specify filenames starting with a hyphen, things would be difficult.
- The order of options relative to one another should not matter (though as the rule below comments, the order of options which require operands might matter).
- The order of operands may matter and position-related interpretations should be determined on a command-specific basis.
- '-' preceded and followed by white space should be used only to mean standard input. (Remember to clear errors and re-open standard input for each one of these; otherwise, your program will only work for the first hyphen.)

In other words, command names are from two to nine lowercase characters or digits long. Options are each a single letter and may be specified as *-x -y* or *-xy*—in any order. Options to arguments must appear as separate arguments and cannot be omitted. All options must precede the operands. The option '--' signals the end of the options. The operand '-' means standard input.

Arguments are passed to the main program through its two parameters:

```
main (argc, argv)
  int argc;          /* number of arguments */
  char *argv[];      /* list of the arguments */
```

The argument *argc* is the count of the number of arguments to the program—including one for the program name itself. Invoking a program with no arguments results in *argc*'s value being 1. The *argv* array contains a list of pointers to character strings, each of which is an argument. Conveniently, *argv[1]* is a pointer to the first argument. Some programs declare *argv* as a double pointer,

```
char **argv;
```

and step through the array one argument at a time.

It is sometimes tricky to write programs which handle the two different argument processing situations:

- The possibly absent arguments represent flags; program input is from *stdin*.
- There are one or more arguments which represent filenames for input.

Here is a standard skeleton from which to begin program construction:

```
/*
 *      Standard argument parsing skeleton
 *      R. Kolstad 2/19/84
 */

main (argc, argv)
int    argc;
char   *argv[];
{
   int     filter = 0;                    /* set if this prog is a filter */

   for (argc--, argv++; argc > 0; argc--, argv++)/* process hyphen params */
   {
        if (argv[0][0] == '-')            /* if '-', process as switch */
        {
           printf ("Switch is %c\n", argv[0][1]);/* process switch */
        }
        else
           break;
   }

/*
 * now we're done and either (a) argc == 0 [filter] or (b) argc > 0
 * meaning that there is a set of files to process
 */

   filter = argc == 0;
   if (filter)
        argc = 1;                         /* filter lets for loop run once */

   for (; argc > 0; argc--, argv++)       /* for each argument */
   {
        if (filter)
           printf ("Process as program is filter\n");
        else
           printf ("Process argument as parameter = %s\n", *argv);
   }
}
```

Keep two points in mind while filling in this skeleton. First, it is handy to be able to specify standard input in an argument list of file names (this is usually done by an argument of '-'). Second, be sure to reset any previous end-of-file indications when using the *stdin* variable in a program which may see multiple standard input files.

The *getopt* package, if available, provides an alternate scheme for parsing arguments. The public domain version was written by Henry Spencer at the University of Toronto school of Zoology. It is almost identical to the Bell version and has been published on net.sources. See its manual section for more details.

Environment Variables

Some programs customize their behavior for users depending on the environment passed to the program. This environment appears in a variable declared

```
extern char **environ;
```

though routines to access this variable include its declaration and usually eliminate the need to declare it. It is easy to discern values of these parameters; the *getenv* routine (see *getenv*(3)) returns the values. Here's a sample program and its output:

```
% cat a.c
char   *vars[] =
{
   "EDITOR", "HOME", "TERM", "SHELL", "USER", (char *) 0
};
main ()
{
   char   *getenv (), **p, *result;
   for (p = vars; *p; p++)
   {
         result = getenv (*p);
         if (result)
            printf ("%s is '%s'\n", *p, result);
         else
            printf ("%s not found.\n", *p);
   }
}
% a.out
EDITOR is '/usr/bin/xed'
HOME is '/mnt/kolstad'
TERM is 'vt100n'
SHELL is '/bin/csh'
USER is 'kolstad'
%
```

If your program is enhanced by some customization, consider the easy use of environment variables as a way to parameterize it. If the program must use a standard variable (such as an editor), try using the available environment variables to give users the uniform treatment they expect. See Chapter 3 for other instances of programs which use *shell* environments.

Initialization Files

Finally, some programs (e.g., *ex* and *mail*) have large numbers of parameters. These programs allow the user to set parameters during interactive use of the program. They also have a standard file which is read by the program just after it is invoked. This file almost always lives in the user's home directory and is typically named *.xxxrc*, where *xxx* is the program's name (e.g., *.exrc* and *.mailrc*). See each program's manual for particular parameters which may make the program more usable to you. Also consider use of such files in any program you write which may have large numbers of user-settable parameters.

Debugging UNIX Programs

It is unfortunate that the two most common runtime error messages produced by the UNIX system are: "Segmentation Fault" and "Bus Error." Uninformative as these sound, both are caused by attempting to reference a memory location that should not be referenced by a user program. This can happen by oversubscripting an array or by dereferencing an invalid pointer. Usually these references are for data reads and writes. Particularly unlucky programs will attempt to invoke a function found in an array of pointers to functions, only to find control being passed to a location preferred by the operating system to be a disk controller register.

There are several ways to produce programs that will guard against these kinds of errors. This section details assertions, debug printouts, printing informative error messages, the use of subscripts and pointers, handling dynamic memory allocation, creation of lists and loops to

process them, and the general trouble of signed characters and integers.

Assertions

Many programmers can make programs faster and more concise by making certain assumptions about the programs or their data. Usually, these programs fail catastrophically if the assumptions turn out false. Inserting assertions into code is one way to ensure that such conditions are met. These assertions might be specified with a *define* statement like this one:

```
#define        ASSERT (boolean, message)  if (!(boolean))      \
        printf("Programname: assertion %s failed, %s\n", \
        "boolean", message)
```

Whenever a program makes an important assumption, insert an *ASSERT* statement. Here is a sample *ASSERT* which aids in detecting division by 0:

```
ASSERT (divisor != 0, "modulename: Result of test must not be 0");
```

Using the correct *Programname* and *modulename* will improve readability of the error messages. See the next section on "Debug Prints," for detailed ideas on the format of debug printouts.

Probably the most ignored source of possible errors is ignoring the return value of system calls. Most system calls can fail in some way or another but do so only rarely. These rare instances can be disastrous. Checking their results can avert real problems. Here is a define statement that might help:

```
#define        CHECK (syscall, message) { if( (syscall) < 0) {perror("CHECK"); \
        printf ("programname: %s", message); }
...
        CHECK ( read(0, line, 80), "Cannot get user's address");
```

This macro checks the results of each system operation and prints the message and the system error string which decodes the value of the global variable *errno*. You can also use a *CHECK* macro which resembles the *ASSERT* macro by explicitly specifying the error conditions. The call to *perror* precedes the *printf* since the global error value (*errno*) may be modified by the *printf*. An alternative is to save and restore the value of *errno*.

Debug Prints

Using small modules coupled with makefiles and fast compilers makes printing debugging information particularly handy on UNIX. Unfortunately, in a larger program, this can get out of hand as the volume of printing increases. It can become difficult to know which module is issuing the debugging information; there can be so many debug prints that it is impossible to see the significant ones. Most UNIX utilities now print the name under which they were invoked as part of their error message. This is handy when error messages occur during a piped sequence of programs.

When debugging UNIX programs, it can also help to include the function name or the module's filename and line number of the debugging print. The module's filename and line number are available as the preprocessor variables *__FILE__* and *__LINE__*, respectively. The variable *__FILE__* is replaced by the current filename surrounded by double quotes; the variable *__LINE__* is replaced by the integer representing the line number. These allow very precise error messages:

```
printf ("Progname: %s line %d: Debug value is %d\n", __FILE__, \
        __LINE__, debugvalue);
```

While it is difficult to envision "too much debugging printout," in practice it can be difficult to

wade through irrelevant sections while searching for an important line. The use of a global debugging parameter can solve this problem by allowing debugging statements to be enabled and disabled at runtime. The technique is simple:

```
long debug;             /* global debug parameter */

#define         MAINDEBUG0x000000001
#define         FILEDEBUG  0x000000002
#define         SORTDEBUG          0x000000004
  :
  :
        if (debug & FILEDEBUG)
                printf("open: Current open files are ...", ...);
```

The dynamic value of the *debug* variable enables and disables all prints dependent on it. Using a 32-bit variable allows 32 switches to be contained in a single variable (more than one variable can be used in an extended scheme). The inclusion of a simple way to set the *debug* variable (some ways are: as a parameter, as an environment variable, in the program's *xxxrc* start-up file, via a debugger) makes this an easy-to-use and powerful technique to use for large programs.

Another technique to control the level of debug printout is to declare a global *debug* variable which is compared to the level given with each debug print. When the global's value is large enough, the print succeeds.

Subscripts and Pointers

The C language contains the familiar array aggregate type. Arrays can be used to allocate a collection of items to another data type. This allocation places the array's elements contiguously through memory, with each element placed at a memory location in a strict linear progression. You can access these elements either by subscripting or by direct reference to their address. Using any kind of element's address usually involves using "pointers."

Pointers make up a highly efficient programming construct which provide a convenient abstraction for dealing with certain kinds of program structures. Consider a list of 25 people's names and phone numbers. A FORTRAN programmer might choose the following data structures:

```
#define         NPEOPLE    25
        int phone[NPEOPLE];
        char *name[NPEOPLE];      /* array of pointers to names */
```

The FORTRAN programmer would then use a person's identification number to subscript these two arrays. If they were to be written to disk, the programmer might create a loop which printed them out some other way: name, phone, name, phone, etc.

The C programmer might see things differently and reason that the name and phone number are part of a common structure for an individual and declare an array of such structures as follows:

```
#define         NPEOPLE    25

        struct person {
                char *name;
                int phone;
        } people[NPEOPLE];
```

While approximately the same amount of storage is allocated, the name pointers and phone numbers are now adjacent to each other in memory. This declaration allocates 25 sets of 2 words each instead of 2 separate arrays. It is now possible to refer to person 7's complete record as

people[7] and his/her phone number as people[7].phone. The convenience comes in passing person 7 as a parameter to a routine. By determining the address of person 7's record (known in C as &people[7]), a routine can have a simple parameter, 'p', which allows convenient notation and efficient code for reaching person 7:

```
printaperson (p)
   struct person *p
{
        printf("%s's phone is %d\n", p -> name, p -> phone);
}
```

This function makes no assumption about the storage technique for the aggregate of people (which may be in an array, queue, or linked list) but accesses person *p*'s data in a handy way. See the "The C Programmer's Manual" by Kernighan and Ritchie for more information.

Dynamic Memory Allocation

The previous example mentioned that storage need not necessarily be limited to arrays when programming in C. By calling *malloc*, programs can ask for memory which can be used to store any kind of data. This storage can then be connected into a linked list or other data structure. The following segment allocates a new slot for a person's name and phone number. It then copies the person's name into new storage after setting up space for the new structure. Finally, it returns a pointer to the new *person* entry.

```
struct person *addaperson(newname, newphone)
char *newname;
{
   struct person *s;
   char *p = malloc (strlen (newname) + 1);/* room for name and \0 */
   if (p == 0) {
        printf("malloc failed ... addaperson (1)\n");
        exit (1);
   }
   s = malloc (sizeof (struct person));
   if (s == 0) {
        printf("malloc failed ... addaperson (2)\n");
        exit (1);
   }
   s.name = p;
   while (*p++ = *newname++);
   return s;
}
```

This routine uses a standard C idiom in the *while* loop. The loop copies the characters from *newname* to *p* until a null character is encountered (the null is copied, too). It is quite permissible to destroy the argument *newname*, though one should be careful when doing so. Note carefully the spaces around the = tokens: the statement "a=*b" parses to "a =* b" instead of the (probably desired) "a = *b". See *malloc*(2) for more information on storage allocation.

The standard I/O package uses *malloc* and *free*. Beware of this when using other storage allocation schemes.

Processing Lists

Programmers often find themselves processing lists of items. The program which prints environment variables illustrates this:

```
char  *vars[] =
{
   "EDITOR", "HOME", "TERM", "SHELL", "USER", 0
};
main ()
{
   char  *getenv (), **p;
   for (p = vars; *p; p++)
        printf ("%s is '%s'\n", *p, getenv (*p));
                                   /* note call to getenv */
}
```

This program uses a complicated pointer scheme (an array of pointers to character strings) and a trick for processing a list. The list is traversed by the *for* loop which specifies three expressions:

1. The first names the list's starting point (in this case, *p = vars*).

2. The second gives a rule which determines whether the traversal is complete. In this case, the rule is **p*. The expression evaluates to the address of the first character. A zero value for this address terminates the list. Coding **p* is the same as coding **p != 0*.

3. Finally, a rule for getting to the next element (in this case, *p++* since *p* was an array; *p = p -> next* is typical for linked lists).

Specifying the zero value as the last element in the list avoids problems with counting elements (both for traversal and for allocation).

Integers and Signed Characters

The processing of character data occasionally leads to problems due to the requirement for a sentinel to appear in the data (such as end-of-file). Because there are only 256 characters and all of them are possible values, it can become cumbersome to deal with another variable that indicates special conditions.

The use of integers to contain characters is the most effective solution to this dilemma. It is quite common to see the following code:

```
#include <stdio.h>
int c;  /* c is an INTEGER */
        ...
        while( (c = getchar ()) != EOF)
        {
                /* process c */
        }
```

The *EOF* token yields the numerical value -1. If *c* were declared to be a character and were assigned the octal value 0377 (a perfectly valid character), then the *while* loop would terminate: 0377 sign extends to -1 on many computers.

Be aware of this kind of problem when coding assignments between variables of type character and those of type integer.

Miscellaneous Conventions

Most of the examples shown in this chapter exhibit a number of C programming conventions. Among these conventions is the style of naming variables. The use of *argc* and *argv* is standard. The names of integer counters are almost always drawn from: *i, j, k, l, m, n*. Character pointers are commonly called *p* or *q*. Structure pointers are often called *s* or *t*, or sometimes *p*.

The use of the *NULL* definition is common. Normally, *NULL* is a 0 value. Unfortunately, the type of the 0 is not always understood well. Occasionally, the *NULL* will have to be cast to a different type for *lint* to be satisfied that all is well.

Symbolic Debugging

The saving grace of the "Segmentation Fault" and "Bus Error" detection mechanisms is the production of a file named *core* in the working directory where the program failed. This *core* file contains the state of the program when it failed and includes registers, global variables, and local variables for active routines. Two debuggers deal with these files: *adb* and *csd*. *adb* is used for programs which have not been compiled and linked with the *-g* option. *Csd* is used for specially compiled and linked programs. All programs contain symbol tables of global variables and routines (see *nm*(1) for a program which will list the names). The debuggers use the tables to find the location of program modules and variables. The additional information and routines required for *csd* make it profitable to use the *-g* option only when debugging is known to be necessary. However, do use it when debugging is required: *csd* is wonderful.

adb

No matter how it is compiled or linked, you can invoke the *adb* debugger for any program. However, if the object code has been "stripped" (see *strip*(1)), *adb* cannot help much. Probably the most valuable information provided by *adb* is the stack dump it provides. Invoke *adb* and request a dump like this:

```
% adb brokenprogram
$c
.... <- stack dump appears here on many lines
```

Examine global variable values by entering their name, a slash, and an output format (usually *X* for 32-bit hex). Here is an example which illustrates debugging the running operating system:

```
% adb /vmunix /dev/kmem
Convex Debugger ($Date: 85/02/08 16:05:13$)

(adb) maxmem/w
_maxmem: 1392
```

Other formats (see *adb*(1)) allow easy printing of other data representations (e.g., characters, strings, and decimal integers); *adb* complexities allow the printing of structure values, complete with labels. Since the advent of *csd*, *adb*'s utility is limited.

csd

The *csd* debugger is a full-fledged source-level debugger (refer to the *CONVEX Consultant User's Guide* for information on the *csd* debugger). In addition to containing a complete machine-level debugger similar to *adb*, *csd* can:

- Run programs (complete with arguments and file redirection)
- Trace lines, procedure calls, functions, expressions, and variables

- Set and delete (possibly conditional) breakpoints
- Single-step program lines (including stepping "through" function calls)
- Print global and local variable names
- Invoke functions from the debugger
- List the source file lines

csd is easy to use (see *csd*(1)) if care is taken to reference local variable names as *file name:functionname.variable* and a *.csdinit* file with lots of *alias* command supplements the debugger itself. Set up a small program and try it.

Capability Files

Bill Joy (formerly of Berkeley, now at Sun Microsystems) devised a scheme for his editor (*vi*) to utilize cursor positioning schemes for many different types of terminals. He created an ASCII file of names and strings which describe the terminal's capabilities. Each name represents a certain capability for a terminal; the string represents the scheme for implementing that capability. The names are usually two letters long. Here is a sample from the file for terminal capabilities known as termcap:

```
Mu|sun|SUN microsystems inc:\
        :li#34:co#80:cl=^L:cm=\E[%i%d;%dH:nd=\E[C:up=\E[A:\
        :am:bs:mi:ms:pt:\
        :ce=\E[K:cd=\E[J:so=\E[7m:se=\E[m:\
        :kd=\E[B:kl=\E[D:ku=\E[A:kr=\E[C:kh=\E[H:\
        :k1=\EOP:k2=\EOQ:k3=\EOR:k4=\EOS:\
        :al=\E[L:dl=\E[M:im=:ei=:ic=\E[@:dc=\E[P:
```

The backslashes precede characters to be output as control characters, as well as indicate continued lines. Percent signs indicate the need for special processing.

The termcap file is an instance of a general class of files known as "capability files." These files include parameters for more complex programs which typically must interface to real-world entities for which there are no standards. The 4.2BSD UNIX system includes a printcap file in addition to termcap. Use capability files to increase flexibility and modularity of your own applications.

File Types

Normal UNIX disk files are typeless. Their attributes include only the file's length and its contents. The operating system kernel does not support special types for fixed-width text, partitioned data sets, or random-access files; these special types are supported by user programs. Because of the buffering scheme used by the operating system, it is not particularly uneconomical to open and read disk files.

For these reasons, it is usually handiest to store data as normal ASCII text instead of binary data. There is a small penalty in reconversion of the data from ASCII to binary form (e.g., for numbers). This cost is mitigated by the great effort required to debug programs which use special file formats and create special editors to manipulate the files' contents. Only for large applications is it profitable to store data in binary format. The "capability files" mentioned previously are examples of this philosophy. (Though, because the number of terminal types has grown so quickly, termcap files will probably be compiled from the ASCII text in the foreseeable future).

The availability of *printf* and *scanf* means that it is usually easy to convert between internal and external representations of files. Do not fear using termcap style output or files which resemble those created by the *mail* programs (i.e., with line prefixes like From: and Date:). Bear in mind that the cost of storing a newline character is the same as any other character. Storing many

short lines requires the same amount of disk storage as fewer, longer lines.

Screen Manipulation Tools

The advent of capability files and the screen editor heralded a new era for the creation of screen manipulation programs. Two packages now aid screen manipulation: "curses" and "Maryland Windows". These packages greatly reduce the amount of time required to construct screen oriented applications. These applications prefer to treat the terminal as a two-dimension array of characters which is updated rather than a "glass teletype." Briefly, these packages each provide a set of routines for writing on a memory-resident image of a screen along with a routine which will update the screen without completely "repainting" it. The "curses" package is well documented, well debugged, and supported. The "Maryland Windows" package is not as popular since it is newer. It is, however, a significantly faster performer than "curses," using less CPU time and finding much better ways to refresh the screen. See the appropriate documentation for descriptions and examples of using these routines.

4

Accelerating Programs

Typically, once a program has been thoroughly debugged, tested, and documented, people begin using it and complaining about its speed. This section describes three ways to decrease a program's execution time: profiling, optimizing and recoding in assembly language.

Profiling

Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick wrote *gprof*, a program which analyzes the CPU consumption of different parts of a program (reference the *CONVEX Consultant User's Guide*). This "profiler" consumes only a small amount of additional memory and execution time to create a local file summarizing CPU time distribution. Obtain profiling information by compiling and linking sources with the *-pg* option. Invoke *gprof* on the monitor file, gmon.out. Routines which consume more than 10% of the CPU time are good candidates for hand optimization or new algorithms. Little is to be gained by making those routines which consume small percentages of the total time more efficient. See *gprog(1)* for additional details.

Optimizing

Mechanical Optimization

You can gain a quick execution speedup by invoking the C optimizer: include the *-O* option on all *cc* invocations. The increased compile time can be gained back in the first 10 CPU minutes of execution for a moderate-sized package; execution time savings of 30% are not unusual. Be advised that the optimization pass will not ease debugging using low- or high-level debuggers.

Hand Optimization

The ultimate savings in time can be had with the ultimate expense in maintenance and program creation: coding of small (CPU intensive) functions and routines in assembly language. This technique is recommended so rarely that it should probably be avoided: programs with such tight speed requirements probably need a faster processor soon, anyway.

5

Advanced Problem-Solving Techniques

Creative Plagiarism

Perhaps the easiest way to construct software solutions is to reuse source code from other applications. The UNIX operating system and utility source code is a gold mine of such applications (though available only to those with a UNIX source license). Locally developed programs often solve problems related to new requirements. Viewing a new problem and its solution in terms of old problems and their solutions can dramatically reduce programming time.

Configuration Programs

Some programs require many parameters to specify the desired solution. An extreme example is the assembly language program generator known as the "C Compiler": its parameters include the source program which describes the function of the generated assembly language.

Certain applications may find it useful to create a sort of meta-program which generates other programs or makefiles as its output. A certain test suite for the C programming language exemplifies this technique. It contains a program generator which creates statements to test the cross-product of all possible operators.

The incidence of a large number of very similar submodules in a project indicates that a module or program generator might reduce development time.

Creative Editor Use

Most UNIX editors match regular expressions during searching or substitution. These expressions following simple rules (see *ed*(1) for a more detailed explanation):

- Characters (except special characters) match themselves.
- The dot . matches any character.
- The backslash \ followed by a character matches that character, unless it is one of (,), or a digit.
- A string surrounded by brackets ([and]) matches any character within the string. If the first character of the string is a caret ^, the bracketed string matches any character not found between the brackets. Backslashes have no special meaning;] can be entered as the second character if it is to be matched. Sometimes the notation a-b will match characters between a and b, inclusive.
- Succeeding a regular expression by a * causes it to match 0 or more occurrences of the expression.
- Bracketing expressions with \(and \) has no effect on matching but does provide a reference to the string matched: it can later be denoted \1, \2, etc., where the digit denotes the number of the bracketed expression as seen from left to right.

- Concatenating regular expressions leads to expressions which attempt to maximize the matching of the first components.
- The ^ character matches the start of a line.
- The $ character matches the end of a line.

Here is a command which reverses the first two words on each line of the file:

```
g/^\([^ ]*\) \([^ ]\)*/s//\2 \1/
```

The bracketed expressions match the first two words and assign them the internal names \1 and \2. The substitution replaces them with their reverse.

It is often possible to edit the output of the *ls* command and create scripts for the shell which manipulate sets of files into a desired form with only a little typing.

File Manipulation

The UNIX utilities include many programs for file manipulation. This section briefly outlines each one, giving occasional counterintuitive applications if applicable. Most of these commands operate on the standard UNIX text files (which have no particular attributes other than their length and contents). Most UNIX files contain variable-length lines terminated by the newline character (octal 012). The *CONVEX UNIX Programmer's Manual* contains detailed descriptions of each of these commands.

sort Orders the lines of a file. It can use any "field" as a key, or start sorting at any column. It can ignore case differences; it can sort in ascending or descending order.

join Examines the contents of the files presented as its input arguments, then concatenates and prints those lines which share a common field. It can extract lines from one file based on those in another, or display those lines which do not match appropriately.

uniq Eliminates those lines of a file which are duplicated: it is useful for maintaining lists in which an item need appear only once but might be input as a duplicate. The *uniq* can also count the number of times each of its input lines appears.

split Breaks a long file into a set of smaller files of any length (expressed in lines). The *cat* command reverses this operation.

diff Produces several different formats which express the "minimal" differences between two files. It can produce an editor script to change one file into another. The utility *diff3* generalizes *diff* to work with three files. It can produce a script which incorporates changes from two evolution lines of a common ancestor into a file which merges those changes.

head Shows the first few lines of a file. *Few* is a parameter.

tail Shows the last few lines of a file. *Few* is a parameter.

cmp Tells if two files are identical or not. Its *-l* switch reports all differences, a single character at a time.

od (Octal dump) Displays a file in any of a number of formats, including octal characters, decimal integers, and hexadecimal words.

tar (Tape archive) Allows you to combine several files into a single file. This single file can then be manipulated with commands that work on single files (like *cp* and *mv*). *Tar* can then extract the files from the combined archive.

Miscellaneous Tools

Local users and those on the network develop widely needed tools. Some that are available in CONVEX's /usr/local/bin directory include:

- *bban* — prints signs on the line printer — a "better banner" program.
- *calls* — does calling tree analysis for C programs.
- *dirtree* and *tree* — print directory trees in both vertical and horizontal formats.
- *tpar* — allows multiple *tar* files to be stored on a tape — keeps a disk-resident directory of the tapes under its management.
- *shar* — takes a list of files and compresses them into a single file which can be transmitted over normal ASCII communications paths (e.g., via UUCP or mail) and later "fed" to a (Bourne) shell for uncompression.
- *ansiread* and *ansitar* — manipulate ANSI standard tapes.
- *taperead* — reads virtually any tape and stores it on the disk.
- *readtape* and *writetape* — copy tapes (by storing original tape on disk) — only one tape drive required.

Manuals for each of these programs are available in CONVEX's local manual directory.

Other High-Level Tools

The UNIX system provides several other tools for building sophisticated software systems. The *lex* and *yacc* packages provide a simple way to parse user input when it is often in a form easily modeled by a formal grammar. The *lex/yacc* template from some other project can be used to speed development of this kind of program. Using this method, it is usually possible to write a nontrivial lexical scanner and parser in less than a day. Consider their use for any program which has complex nonnumerical data input requirements.

The *awk* interpreter is a useful tool for dealing with files consisting of data which can be modeled as "fields." It includes associate arrays, typeless arithmetic and character field operations, and C-like control structures. *awk* is useful for quick-and-dirty programs such as those for simple resource accounting. Here's a script used to summarize UUCP logfiles:

```
#
# initialization:
BEGIN   { anytx = 0; anyrc = 0; intotal = 0; outtotal = 0; time = 0; }

# for each line:
   {
      if( $5 == "sent" ) {
         anytx = 1; txbytes[$2] += $7; txtime[$2] += $9;
```

```
            txsystem[$2] = $2; next;
        }
        if( $5 === "received" ) {
            anyrc = 1; rcsystem[$2] = $2; rcbytes[$2] += $7;
            rctime[$2] += $9; next;
        }
    }

# after all lines have been read:
END    {
    if (anytx) {
        print "Data sent";
        printf " system          bytes   time      rate (cps)\n";
        printf "-------------------------------------------\n";
# txsystem is an associative array:
        for ( i in txsystem) {
            t = txtime[i];
            if (t===0) { t=1; }
            n = txbytes[i];
            printf "%-10s %10d ", txsystem[i], n;
            printf "%4d:%02d:%02d ", t/3600, (t%3600)/60,t%60;
            printf "%10.2f\n", n/t;
            intotal += n; intime += t;
        }
        printf "Total sent : %d bytes (%d:%02d:%02d)\n\n", \
            intotal, intime/3600, (intime%3600)/60,    \
            intime%60;

    }
    else {
        printf "No data transmitted\n";
    }
    if (anyrc) {
        print "Data receive";
        printf " system          bytes   time      rate (cps)\n";
        printf "-------------------------------------------\n";
        for ( i in rcsystem) {
            t = rctime[i]; n = rcbytes[i];
            printf "%-10s %10d ", rcsystem[i], n;
            printf "%3d:%02d:%02d ", t/3600, (t%3600)/60, t%60;
            printf "%10.2f\n", n/t;
            outtotal += n; outtime += t;
        }
        printf "Total received : %d bytes (%d:%02d:%02d)\n\n",    \
            outtotal, outtime/3600, (outtime%3600)/60, outtime%60;
    }
    else {
        printf "No data received\n";
    }
    total = intotal + outtotal;
    time = intime + outtime;
    printf "Grand Total : %d bytes (%d:%02d:%02d)\n",
        total, time/3600, (time%3600)/60, time%60;
    }
```

Here's a sample output from this code:

Data Sent System	Bytes	Time	Rate (cps)
ctvax	1205	0:00:09	133.89
uiucdcs	11947	0:01:38	121.91
mostek1	774	0:00:06	129.00
rice	451	0:00:03	150.33
unmvax	569	0:00:04	142.25
allegra	12015	0:01:43	116.65
infoswx	1990	0:00:12	165.83
ihnp4	571	0:00:04	142.75
smu	21036	0:02:54	120.90

Total sent : 50558 bytes (0:06:53)

Data Received System	Bytes	Time	Rate (cps)
ctvax	5052	0:01:01	82.82
uiucdcs	9718	0:01:44	93.44
unmvax	8619	0:01:29	96.84
allegra	76379	0:21:21	59.62
ihnp4	653	0:00:09	72.56
smu	14901	0:02:40	93.13

Total received : 115322 bytes (0:28:24)

Grand Total : 165880 bytes (0:35:17)

The program resembles a C program with no types which is augmented by associative arrays. *awk* enables quick development of programs which use fields.

The stream editor *sed* applies a series of editing commands to a file in a single pass over the file. Its commands resemble those of *ed*(1). Use with scripts when simple transformations are required in batch mode.

The macro processor, *m4*, is an easy-to-use program which substitutes possibly parameterized macros in any text. The manual page (*m4*(1)) gives enough information to get started.

6

Tools for Larger Projects

UNIX provides communications and revision control tools useful both to projects staffed with several people and those that require audit trails. Communications tools include electronic mail and notesfiles (public and private bulletin boards). The Revision Control System (*RCS*) provides version control and file locking for development projects.

Communications

Mail

Electronic mail provides a batched, queued means of sending text to any user or group of users located on any of several thousand different computers throughout the world.

The UNIX mail system can forward mail via an "alias" to one or more users. Replies to a message can be sent to the entire group (via the *r* command) or to just the originator of the message (via the *R* command).

The UNIX mailer often has connections to other UNIX machines. The UUCP network (a superset of the USENET) provides electronic mail access to over 2,100 (as of 2/22/84) machines. Some sites allow simple access to network mail: to mail to "signon kolstad" at the machine named convex, type:

```
% mail convex\!kolstad
```

and enter your mail as usual. The institution initiating the connection between your machine and the network bears the costs of network transactions. It would be an abuse to send over the network inappropriate or verbose messages, but feel free to ask for help or consult companions located on remote machines.

Notesfiles

The *notesfile* system, written by Ray Essick and Rob Kolstad while students at the University of Illinois (distributed by uiucdcs!essick or convex!kolstad), provides computer-managed discussion forums. Discussions can have many different purposes and scopes; usually each "notesfile" discusses a single topic.

Each notesfile contains a list of logically independent notes (called base notes). A note is a block of text with a comment or question intended to be seen by members of the notesfile's community. The *notesfile* system displays the text, its creation time, its title, and the author's name (some notesfiles allow anonymous notes). Each base note can have a number of "responses": replies, retorts, clarifications, further comments, criticism, or related questions that concern the base note. Thus, a notesfile contains an ordered list of ordered lists.

Some companies use notesfiles extensively for communications about projects, general tidbits (e.g., company parties), and discussions of technical and policy decisions.

The *notesfile* system provides a "sequencer" which allows the reader to peruse a selected set of notesfiles, viewing only those notes that are unseen since the reader's last visit. The 'sequencer' allows quick perusing of the latest information about a given topic.

The *notesfile* system also interfaces to USENET, a community of over 700 (as of 2/22/84) UNIX machines. Information from more than 200 notesfiles (also called "newsgroups") is transferred among the machines on a timely basis. It is quite possible to suffer from information overload by trying to keep up with too many networked topics.

Pitfalls

Relying on electronic communication can have its disadvantages. You do not always know when the mail to a remote computer is received, for example. Electronic mail and notesfiles both suffer due to the frequent failure of the written word to communicate emotional content and subtleties. Sarcasm is almost impossible to convey through a series of characters on a CRT screen. Worse still, using a keyset can cause people to forget that the person reading their message on the other end has feelings and emotions, unlike the terminal. It is suggested that electronic communications users choose their words carefully to avoid insulting or alienating their working partners.

Electronic dialog can be time-consuming and counterproductive. Negotiations and 5-minute discussions are forums probably best served by the telephone or small group meetings. Electronic communications can then be used to disseminate the results.

RCS

Walter Tichy's Revision Control System (*RCS*) (distributed on the 4.2BSD tape) enables version control for any kind of text file on UNIX. The *RCS* stores an annotated copy of each file under its control in a subdirectory (named *RCS*) of that file's directory. This *RCS* file contains both the current file's contents and enough information to generate any "version" of the file that has been checked into the system.

Creating an *RCS* directory is easy, just enter:

mkdir RCS

To check new files into the directory, use the *ci* command:

ci filename

and answer the questions. Each time a file is checked in, data for reconstituting its current state is stored with the data from which *RCS* can conjure any previous checked-in version. Checked-in files disappear from the project directory until they are checked out.

Check out a file with:

co filename

or specify the *-l* switch for locking:

co -l filename

If a file is "locked," no one else may check it out with the *locked* switch until it has been checked back in. This feature allows many developers to work within a large project with the confidence that they will not overwrite other developers' changes.

Files can be checked out for inspection or editing. Checking out a file without locking recreates the file with read-only permissions.

Using *RCS* and the *make* program can be difficult. It is hard to convince *make* that it need not check out a source file to see if its object file is out-of-date. Of course, any time a file is checked out, the file has the current date (rendering any object file out of date).

RCS uses "backward deltas" instead of the "forward deltas" of Bell Laboratory's *SCCS* (which means that the current source is stored in a file along with commands to revise it backward into the past; *SCCS* stores the original source with commands to revise it to the present version). *RCS* is an efficient and easy-to-use package for simple version control tasks.

7

Documentation Hints

Introduction

Documentation provides the final foundation for a properly done software project. UNIX aids documentation production with tools that:

- Ascertain document length.
- Check for simple typographical errors like spelling.
- Analyze writing style and word use.
- Automatically create indexes.
- Aid in efficient text processing.

Wc

The word count (*wc*) program tallies the number of lines, characters, and words found in a document. Here is a script of a session which invokes *wc* on one version of this chapter:

```
% wc chapter8
   134     848    5094 chapter8
```

This chapter has 134 lines, 848 words, and 5094 characters.

Spell

The *spell* program points out typographical errors. Some sites augment spell with a *sp* shell script. For each of its arguments, the *sp* script creates a file of potentially misspelled words. When using *sp*, file *foo*'s tentative misspellings appear in file *sp.foo*.

Diction/Style

The CONVEX implementation of Bell Laboratory's "Writer's Workbench" includes the *style* and *diction* programs. These two writing aids summarize writing flaws and hence give nonhuman constructive criticism to writers about their text.

Diction

The *diction* program searches text for cliched or grammatically incorrect constructions and displays the phrase in context. The *explain* program suggests alternative wording for *diction*'s complaints. Here's an example:

```
% diction chapter8
chapter8
Some sites provide the  sp  shell script *[ which ]* takes as
its argument a list of text files and creates a list of possibly
misspelled words in each file.
number of sentences 9 number of phrases found 1
```

```
% explain
phrase?
which
use " "that" when clause is restrictive" for "which"
use "when" for "at which time"
```

Style

The *style* program makes analyses of various parameters of its input: sentence length, sentence type, word length, and so on. Here is a sample of its output on the paragraph about the *spell* program above (lines starting with #) are explanations of the output:

```
readability grades:
        (Kincaid) 7.6 (auto) 8.5 (Coleman-Liau) 9.9 (Flesch) 8.2 (68.4)
# these three different indicators represent what grade level of reading
# is required to understand the passage and range from grade 7.6 through
# almost the tenth grade. The number 68.4 can range from 1 ("Run, Spot,
# Run.") to 100 (complex biological Ph.D. theses) and indicates approximate
# complexity of the sentences.

sentence info:
        no. sent 3 no. wds 47
        av sent leng 15.7 av word leng 4.68
        no. questions 0 no. imperatives 0
# self-explanatory

        no. nonfunc wds 32  68.1%   av leng 5.63
# nonfunction words are nouns, adjectives, adverbs, and nonauxiliary
# verbs. Their average length may be more useful than the overall average
# word length
        short sent (<11) 33% (1) long sent (>26)  33% (1)
        longest sent 28 wds at sent 2; shortest sent 8 wds at sent 1
# certainly want to vary those sentence lengths to avoid monotony

sentence types:
        simple  33% (1) complex  67% (2)
        compound  0% (0) compound-complex  0% (0)
# self-explanatory

word usage:
        verb types as % of total verbs
        tobe  0% (0) aux  25% (1) inf  0% (0)
        passives as % of non-inf verbs  0% (0)
        types as % of total
        prep 10.6% (5) conj 2.1% (1) adv 4.3% (2)
        noun 34.0% (16) adj 21.3% (10) pron 2.1% (1)
        nominalizations  2 % (1)
```

```
# self-explanatory

sentence beginnings:
        subject opener: noun (0) pron (0) pos (0) adj (1) art (1) tot  67%
        prep   0% (0) adv   0% (0)
        verb   0% (0)  sub_conj  33% (1) conj   0% (0)
        expletives   0% (0)
# self-explanatory
```

See the document "Writing Tools--The STYLE and DICTION Programs," by L. L. Cherry and W. Vesterman in the *CONVEX UNIX Tutorial Papers* for more details.

Indexes

Computer manuals often lack meaningful indexes. It is simple to construct index macros when making a document. Take advantage of them and make your document useful, too. The macros used to write this text include the *.IN* construct which notes the specified phrase and page number for later inclusion in an index. Here's a sample invocation of an index macro written in *troff*:

```
.IN "Indexes"
```

Here's the *.IN* macro:

```
.de IN
\&.tm \ \\$1~\\*A~\\n%
..
```

The string contained in register *A* contains the current chapter number or name.

Store the output of *troff*'s *.tm* commands by redirecting standard error output into a file:

```
itroff -mxx file list >& index.out
```

The *runindex.t* program extracts the text between the markers and prints an index.

Manual Pages

In many shops, a "man" page must be produced for every package that ever leaves a programmer's directory. Manual pages take only a few minutes to prepare and considerably enhance the program's usability. It also makes it easy to ship the program to others who need it. Find a similar program, copy its manual page to a temporary buffer, and modify it to suit your own purpose.

Efficient Text Processing

The *nroff* and *troff* programs can consume large amounts of CPU time in a text-processing environment. Users often want to rerun a document to get "the last typo out" and hence expend much CPU time for a small fix. A couple of suggestions are to strive for excellence rather than perfection, and to run chapters singly until the document is ready to be integrated.

When using *nroff*, fixed-length output pages ease some problems. The *getpg* program gleans fixed-length pages from *nroff* output. Its invocation consists of its name, the output file, and a list of pages to retrieve:

getpg outfile 4 1-3 9

getpg's output is in page order, no matter what its input. *getpg* allows the printing of just a few changed pages of a document.

8

Conclusion

The UNIX system provides a powerful framework for developing and prototyping software systems. The use of modular software, exploitation of existing tools, and outright plagiarism can shrink development time and aid production of systems which are useful in many different contexts.

This document has described various techniques used by programmers in some shops to increase their productivity. Feel free to develop other methodologies. Post them to the network as you discover them in order to continue the sharing and growth that has characterized UNIX development in the past.

A

Problem Reporting

Introduction

The *contact* utility is the recommended way to report software and documentation problems to the Technical Assistance Center (TAC). It is an interactive tool that prompts you for the information necessary to report a problem to the TAC.

You must have a UNIX-to-UNIX Communications Protocol UUCP connection to the TAC to use *contact*. A UUCP system allows communication between UNIX systems by either dial-up or hard-wired communication lines. See *uucp*(1) or the entry in *info*(1) (online information system) for more information.

Information Required to Report a Problem

contact requires the following information:

1. Your name, title, phone number, and corporate name.

2. The name and version of the product involved. If you do not know the version number of the program or utility you are having trouble with, use the *vers* command:

   ```
   % vers /usr/convex/fc
   /usr/convex/fc: 2.1
   %
   ```

   Be sure to use the full path name of the program when invoking this command. If you don't know the full path name of the program, you can use the *which*(1) command.

   For more information on the *vers* command, see *vers*(1) in the *CONVEX UNIX Programmer's Manual*, Part I.

3. A short (1 line) summary of the problem.

4. A detailed description of the problem. Include source code and a stack backtrace whenever possible. (See *adb*(1) or *csd*(1) for information on obtaining stack backtraces.) The more information provided, the quicker your problem can be isolated and solved.

5. The priority of the problem. You are shown a list of six levels from which to select.

6. Instructions for how to reproduce the problem, including the command syntax used, any flags invoked, or any other things you attempted to do to make your program run.

7. Any other comments you have about the problem or files you wish to submit.

You will have a chance to review your report before you submit it. You can edit the report if you find an error in what you have typed. If you change your mind and don't want to submit the report, you can abort the *contact* session; the file is saved in your home directory in a file named *dead.report*.

The following figure is a sample *contact* session.

Figure A–1: Sample *contact* Session

```
%contact RETURN
Welcome to contact version 0.7 (86/04/01)

Enter your name, title, phone number, and corporate name
(^D to terminate)
> Margaret Atwood, systems programmer, 814-4444, University RETURN
> of Chicago RETURN
> CTRL-D

Enter the name and version of the product involved
> CONVEX UNIX Programmer's Manual, Part I, Revision 4.0 RETURN

Enter a short (1 line) summary of the problem
> The finger command manual page lists nonexistent bug RETURN

Enter a detailed description of the problem (^D to terminate)
> The finger(1) man page says, under the BUGS section, that RETURN
> "Only the first line of the .project file is printed." RETURN
> Happily, this is not true! RETURN
> CTRL-D

Enter a problem priority, based on the following:
1) Critical    - work cannot proceed until the problem
                 is resolved.
2) Serious     - work can proceed around the problem,
                 with difficulty.
3) Necessary   - problem has to be fixed.
4) Annoying    - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
> 4

Enter the instructions by which the problem may be reproduced
(^D to terminate)
> a) put more than one line in .project RETURN
> b) read the man page for finger(1) RETURN
> CTRL-D

Enter any comments that are applicable (^D to terminate)
> The man page should be updated. RETURN
> CTRL-D

Are there any files that should be included in this
report (yes | no)?
> no RETURN

Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
> 3 RETURN

Problem report submitted.
```

%

Index

U

V

W

Y

Software Documentation

Index Enhancements

So that we can continue to provide better indexing in CONVEX documentation, please keep track of the words or phrases you look up in an index, but don't find. Then, list under which index entry you ultimately found the information you were seeking. You can mail one of these postage-paid forms to the CONVEX Software Documentation Department monthly, or you can submit the information to the Technical Assistance Center in the form of a bug report. You can get more forms by writing to CONVEX at the address below, or by calling us. You can also photocopy this form and mail it back in an envelope. Thank you for helping us to serve you better.

Name:_____ **Company:**_____

Phone:_____ **Date:**_____

Manual Title/Rev. No.	Looked Up This Word	Found Information Under This Word

CONVEX Computer Corporation — P.O. Box 833851, Richardson, TX 75083-3851 (214) 952-0200

(Fold Here First)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CUSTOMER SERVICE
CONVEX Computer Corp.
P.O. Box 833851
Richardson, TX 75083-3851

(Fold Here Second)

(Tape or Staple)

CONVEX Guide to Software Development
Document No. 710-001930-200, Version 3.0

Reader's Forum

You are invited to submit your comments concerning the clarity and service of this manual. Constructive critical comments are most welcome and will help us continue in our efforts to generate quality customer documentation. (Please list the page number for questions, comments, and errors):

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

From:

Name_____ Title_____

Company_____ Date_____

Address_____

FOR ADDITIONAL INFORMATION/DOCUMENTATION:

In the continental USA, please call (214) 952-0200.
In Europe, please call 011-44-483-69000.

Direct Mail Orders to:

CONVEX Computer Corporation
Customer Service
Educational Department
P.O. Box 833851
Richardson, Texas 75083

(Fold Here First)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CUSTOMER SERVICE
CONVEX Computer Corp.
P.O. Box 833851
Richardson, TX 75083-3851

(Fold Here Second)

(Tape or Staple)