The Software Toolworks®

14478 GLORIETTA DRIVE, SHERMAN OAKS, CALIFORNIA 91423 (818) 986-4885

C/NIX Version 1.56
User's Manual
January 1985

S. Tucker Taft
C/Craft
22 Downing Road
Lexington, MA 02173

CONTENTS

Introduction to C/NIX 3
How to Get Started 3
Organization of this Manual 3
Where to Begin Reading 4
Problems and Comments 4

General Information Chapter (info) 5
intro - Introduction to C/NIX Features 5
Making Backup Distribution Disks 5
Running C/NIX (The Quick Way) 6
Installing C/NIX 7
Advanced Installation Notes 8
Your First Use of C/NIX 8
How to Continue 9
Notes for CP/M Users 10
Hierarchical Directories 10
I/O Redirection and Pipes 11
The Rest 12
ioredir - Input/Output Redirection and Logging 13
patches - Patches to the C/NIX System 16
pathname - Hierarchical Directory Pathname Syntax .. 18
pipes - C/NIX Pipes 20
quoting - Quoting and Escape Characters 21
trouble - Trouble Shooting on C/NIX 22
wildcard - Filename Wildcards '?' and '*' 24

C/NIX Commands Chapter (cmnd) 25
commands - Summary of C/NIX Commands 25
cat - Concatenate or Type Files 26
chdir - Changing/Printing Current Working Directory 27
chmod - Change File Mode Flags 28
cp - Copy a File or Files 29
csh - The C/NIX(tm) Shell 30
Command Format and Search Path 30
Command Files 31
Exit Status 33
echo - Echo a string 34
exit - Exit C/NIX Shell 35
grep - Generalized Regular Expression Parser 36
help - Help From the On-Line User's Manual 38
ls - List Directory 40
mkdir - Make and Remove Directories 42
mkrel - Make Page-Relocatable Program 43
mv - Move a File or Files 44
rm - Remove File or Files 45
set - Set Console Output Modes 46
walk - Walk the Directory Tree 47

C/NIX Subroutines Chapter (subr) 48
subrs - C Subroutines for C/NIX 48
bdos - C Interface to BDOS Calls 49
getc - Standard Buffered Character Input 50
fopen - Buffered File Opening and Closing 51
main - Hidden Main Routine; Exit Routine 52
malloc - Dynamic Memory Allocation and Release 53
putc - Standard Buffered Character Output 54
strutils - Standard String Utilities 55

Index 56

Introduction to C/NIX

HOW TO GET STARTED.

C/NIX is a software package which provides your 8-bit CP/M 2.x system with many of the features of UNIX (and MSDOS 2.0). It is intended primarily for use with hard disk or high capacity floppy disk systems, and requires Version 2 of CP/M (not 1.x or 3.x).

In order to use C/NIX, you will first have to install it on your system. Then it will help for you to understand the new C/NIX features you will have at your disposal.

You will not have to read very much of this manual in order to do that. In fact (following an old UNIX tradition), most of this manual is not really meant to be read at all, but simply to be referred to when you need information on a particular command.

However, because these new features involve some complicated concepts like hierarchical file directories and input/output redirection, you should be sure to read carefully the few pages that will help you get started.

Those pages are the very next section, entitled "Introduction to C/NIX Features". We have tried to make this part easy to read. In that section, you will learn how to install and run C/NIX, and how to use some of the features.

The rest of this introduction tells how to find your way about the rest of the manual, and how to report problems to us. If you are impatient to get started, you can skip to the next section now, and return here later.

ORGANIZATION OF THIS MANUAL

This manual is organized following the long-established tradition of UNIX manuals. It's probably different from other manuals you have seen, and it can be very confusing if you try to use it the wrong way.

Except for the very next section, intro, this manual is organized to be referred to, not to be read. You probably won't get very far if you try to read it from cover to cover. The way to use it is to go looking for specific information. In order to do that, you need a general idea of what information it contains, and you need to know how to find that information.

C/NIX contains three kinds of enhancements to your CP/M system. These are features, which you can use in most CP/M commands, new commands to perform some new functions, and subroutines, which are only of use to C programmers.

The C/NIX User's Manual is organized into three Chapters, each dealing with one kind of enhancement: General Information (info), which describes the features, C/NIX Commands (cmnd), and C/NIX Subroutines (subr). In this manual, when one of the enhancements is mentioned by name, you will usually see also the manual chapter in which it can be found: for example, help(cmnd).

Each chapter consists of entries, called pages, each explaining one command or feature. (A manual "page" can in fact take up several screenfuls or paper pages, but it's still called a page.) Pages are in alphabetical order within each chapter.

WHERE TO BEGIN READING

You should begin by reading the remainder of this section. Then read the first (info) page, "Introduction to C/NIX Features", and browse through the rest of the (info) chapter to get an idea of what features are available.

In each of the other two chapters, there is a general introduction page, which gives a bit of a "roadmap" to that chapter's highlights, followed by the remaining pages in alphabetical order.

You should look over the Table of Contents and the introductory page to the Commands chapter to get an idea of what is available, and then read the pages that interest you. If you are a C programmer, you will eventually want to do the same with the Subroutines chapter.

The individual manual pages are organized to contain as much information as possible in a small space. You may find them very terse at first, but you will get used to the presentation, which is designed not only for the printed manual but also for the interactive help(cmnd) facility in C/NIX.

All the manual pages follow certain layout and typography conventions, especially in the SYNOPSIS section which gives a compact but cryptic summary of the command or feature being explained. The notation is explained in the manual page for help(cmnd).

PROBLEMS AND COMMENTS

If you have problems reading your C/NIX distribution disk(s), contact The Software Toolworks. However, due to the large number of features in C/NIX, The Software Toolworks' technical support department may not be able to answer all possible questions about using C/NIX.

A limited amount of telephone assistance is available from C/Craft during the hours shown below. However, it would be appreciated if, before seeking that assistance, you try to exhaust the information provided by the manual, including the table of contents, index, and trouble(info) section.

C/Craft
22 Downing Road
Lexington, MA 02173

(617) 862-8177: 9-10 AM and 8-10 PM Eastern only.

Recommendations for improvements to this manual are welcome, as are all comments on C/NIX. There is a short trouble-shooting guide in trouble(info), which tries to anticipate problems which may come up.

If you think you have really found a bug or undocumented limitation, please make a record of a minimal sequence of operations which illustrates the problem. Include the version number and date displayed when you start up C/NIX. Mail the information to C/Craft at the address above.

Please enclose a self-addressed, stamped envelope if you wish an immediate reply. We can only respond to problems raised by registered owners of C/NIX, so we encourage you to fill out and mail the registration form.

Installing C/NIX

This section describes how to install C/NIX on your system disk. Boot up on your system disk, and use the DIR command to make sure it contains the files stat.com and pip.com. (These files are supplied with your CP/M operating system.)

Find out how much free space you have by typing the command

```
A> stat
```

If you do not have at least 42K of space, you will need to delete files in order to make that much room before you can install C/NIX.

Place the C/NIX distribution disk in a floppy drive (but not the drive you use for your system disk). If your C/NIX system came on more than one distribution disk, use the one marked "Disk 1". We will assume that drive is B, but if your system uses a different drive letter then use that letter instead of B throughout this procedure.

Type the command

```
A> pip a:=b:cnix*.*
```

The names of the three files being copied will print out. When the A> prompt reappears, type

```
A> cnix
```

You should now see something like:

```
C/NIX(TM) SHELL 1.56 12-Jan-85
  Copyright (C) 1985 C/Craft (TM) Lexington, MA

A$                  — C/NIX is awaiting your command
```

You will now copy some of the other C/NIX files onto your system disk, depending on how much space you have on your system disk, and which features of C/NIX you are likely to want to use. The files you will copy have to do with:

The grep command is useful for searching for a word or text string in files. The set command lets you alter some system parameters. To install them, type the command

```
A$ cp /b/grep.pre /b/set.com .
```

(The b in this command is the drive letter for the C/NIX disk, so if your system calls it by a different letter, use that letter instead.)

A powerful feature of C/NIX is the help or man command, which can bring up on the screen any of the information pages contained in this manual. To install the manual page files, you will need about 114K of free space on your system disk. Type the commands

```
A$ mkdir help
A$ cp /b/*.hlp help
```

If your C/NIX system came on more than one distribution disk, replace the disk in drive B by the one marked "Disk 2", and again type the command

```
A$ cp /b/*.hlp help
```

If you have three distribution disks, repeat this step for "Disk 3" as well.

Advanced Installation Notes

This section describes some installation features for the more advanced user. If this is your first time through this manual, you may want to skip this section for now.

Many versions of CP/M allow you to install a command which is executed automatically on "cold boot", when you boot up. Often the program which allows you to specify this command is called CONFIG or CONFIGUR or something like that; consult your CP/M user's manual for details. If your CP/M provides that facility, you can set it to execute the CNIX command automatically so that you always enter CNIX when you boot up.

When C/NIX begins executing, it looks for a file named cnixinit.sub. If this file is found, it is executed as a submit file. This allows you to execute one or more commands automatically whenever you enter C/NIX.

If you are a C programmer, you may wish to copy the C program sources and the mkrel command from the distribution disk(s) onto your system disk for later use. The files to copy are **mkrel.com**, ***.c** and ***.h**. (If there is more than one distribution disk the files may be split among the disks.) You may want to use mkdir(cmnd) to create a subdirectory for the source files.

If space is a problem on your system, you may want to install all the C/NIX files on a disk drive other than the one you boot up on. This will work fine, provided you always run the CNIX command from the current logged in drive. For example, if your C/NIX files are on drive B, make sure the prompt is B: before running CNIX.

You can alter some of the ways C/NIX interacts with the terminal to make the system more to your taste. The system can "page" console output or not, request confirmation or not before overwriting files from certain commands, and print out more details of command files and certain commands as they execute. See set(cmnd) and patches(info) for more details.

Your First Use of C/NIX

Now that you have installed C/NIX, you can run any CP/M command or program, or any of the C/NIX commands. First, you may want to see a list of the files in the C/NIX diskette top-level directory:

```
A$ ls -l /b     (if the C/NIX disk is in B:)
```
or simply:
```
A$ ls -l        (if you installed C/NIX on A:)
```

You should now see something like the following, although the totals will vary according to your disk format and C/NIX version:

```
-rwx  28.6k  cnix.com
-rw-   1.9k  cnixhigh.sys
-rwx  11.0k  cnixutil.pre
-rwx  10.4k  grep.pre
drw- 114.0k  help.sda
-rwx   3.0k  set.com
— Total 139k out of 241k —
```

If you installed the help files on your system disk, you created a sub-directory called help. The next few commands will let you look at files in that sub-directory. For an explanation of directories, filenames and pathnames, see "Hierarchical Directories" below.

Try typing ls -l help, and you will get a listing of the contents of the help sub-directory.

You will notice that the display stops after 23 lines. To continue, type any key on the keyboard. This "paging" of screen output is a feature of C/NIX which prevents information from moving off the screen before you can read it. If you don't like it, you can turn it off; see set(cmnd) or patches(info). However, the help command will always page.

Type out one of the files; for example, try type help/intro.hlp. You should see this file. The help command can also be used to accomplish the same thing. For example, type help intro. help alone is roughly equivalent to ls help; try it.

When you are done with C/NIX, you may type bye or exit, and you will return to CP/M. Alternatively, you may simply power off the computer. When you do turn off the power, it is prudent to first open disk-drive doors to lift the recording heads from the medium.

The next time you boot up your operating system, you can enter C/NIX again by typing the command

```
A> CNIX
```

How to Continue

Now that you have run C/NIX, you may want to learn more about what it can do. The first thing to read is the "Notes for CP/M Users" section below, which explains a number of concepts that are unique to C/NIX.

You can then continue by reading the manual sections on the C/NIX shell (csh(cmnd) or help csh), I/O redirection and pipes (ioredir(info) and pipes(info)), and hierarchical pathnames (pathname(info)). A quick summary of all of the C/NIX commands can be found in commands(cmnd). The notation used in the command summary (SYNOPSIS) on most manual pages is explained in help(cmnd).

NOTES

The diskette from which cnix.com is run must remain on the same drive throughout use of C/NIX. The command interpreter, or shell, always looks to the top-level directory of this drive to reload itself between program executions, as well as for the help sub-directory, and the program cnixutil.pre which contains the commands chmod, mkdir, and rmdir. (The file cnixhigh.sys is loaded into high memory when C/NIX first starts, and then remains resident

until C/NIX exits.)

NOTES FOR CP/M USERS

The C/NIX interface is modelled closely after UNIX. Many UNIX features, like hierarchical directories, pipes, and I/O redirection, are also used in MSDOS 2.0 and higher. A understanding of either of these systems, preferably UNIX, will help greatly in making use of C/NIX.

This section provides a quick introduction to the UNIX-like features of C/NIX. Any of the several books on UNIX would also be a help.

Notes for CP/M Users -- Hierarchical Directories

Probably the most important UNIX-like feature of C/NIX is the hierarchical directory system. This is especially useful in helping you organize your files on a hard disk or other large capacity disk, where a directory listing can go to many screenfuls.

With hierarchical directories, instead of having all of the files in a single directory, you can create a subdirectory, and put a group of files into it, say all the files you need for a particular project. Directories are created using the command **mkdir**.

Having a subdirectory is something like using a separate disk drive to hold just those files, except that instead of drive letters, subdirectories have names, and the files in several subdirectories can go on the same disk. When you do a directory listing in the main directory, all you see is the subdirectory name, not the files it contains.

If you want to further organize the files in the subdirectory, you can create sub-subdirectories within it, and so on. In this way, you get a tree structure of directories on the disk, where the "top-level" directory is the trunk, and each subdirectory is a branch.

The top level directory of a disk is always named by a slash and the disk letter. For example, the top level directory of disk A is always called /a. When you first run C/NIX, you are in the top level directory of the disk with C/NIX on it, usually /a. To see that, you can type the command **pwd** (for "print working directory"), which will show the current directory name.

In CP/M, if you organize your files on different drives, you can move over to other drives by typing, for example, B:. Similarly, you can move to different directories. To move to the directory **help**, type **chdir help**. Chdir means "change directory". To see the name of the new current directory, type **pwd**. This will display the name **/a/help**.

/a/help is an example of a pathname, which is a way to refer to files or directories that might not be in the current directory. A pathname is just a list of subdirectories separated by slashes. The path may start with a slash and a disk letter, which means it starts in the top level directory of a diskette. If it does not start with a slash, it starts from the current directory. For example,

/d/sources/compiler/parser.c

might be the "full pathname" for a file named **parser.c**. The subdirectory

sources is in the top-level directory of disk D. The subdirectory compiler is within sources, and the file parser.c is within compiler. If a directory listing is requested for the (top-level) directory of disk D, only the subdirectory sources will appear. Compiler and parser.c will only appear when a directory listing is requested of their respective parent directories (/d/sources and /d/sources/compiler).

You can use full pathnames in all C/NIX commands where a filename is required. For example, instead of changing directories to type the file parser.c, as in

```
B$ chdir /d/sources/compiler
D$ type parser.c
D$ chdir /b
B$
```

you could simply say

```
B$ type /d/sources/compiler/parser.c
```

Even if you are not in the top level directory, you still have access to files and commands stored there. See "The Rest" below for more details. See also the manual pages for pathname(info), chdir(cmnd), and mkdir(cmnd) for more information on hierarchical directories.

The walk command lets you display all the subdirectories in any directory. It can also perform an operation in each subdirectory. See walk(cmnd).

Notes for CP/M Users -- I/O Redirection and Pipes

Another important UNIX-like feature of C/NIX is I/O redirection. Normally, many commands or programs simply put their output on the console. The ls (or dir) command is one example.

I/O redirection lets you redirect the output of a command or program into a disk file. Or you can provide the input for the program from a file instead of from the keyboard (see ioredir(info)). You can also feed, or pipe, the console output of one program back in as the console input for another program (see pipes(info)).

For example, suppose you want to list all the files in the current directory with the SYS flag set. The ls -l command will list these files with the "rws" flags:

```
-rws   13.2k   clibrary.rel
```

for example. So you want to see just those lines in the output of the ls -l command which have the string "rws" in them.

With C/NIX, you can redirect the output of the ls command to a file. Then you can use the grep command to show you the lines containing the string "rws".

```
$B ls -l >+ tempfile      — save output in tempfile
                          — but also display on console.

... you see the complete ls listing ...

$B grep rws tempfile      — run grep to select "rws" lines.

... you see just the "rws" lines.

$B era tempfile           — and clean up the tempfile.
```

You can do this whole process even easier, in one step, using the C/NIX pipe feature. The command character | will pipe the output of one program into the input of the next. So the single command

```
ls -l | grep rws
```

will pipe the output of **ls -l** through **grep rws**, and you will see all the "rws" lines from **ls -l** in one step.

Notes for CP/M Users — The Rest

Most of the other features of C/NIX should be familiar to CP/M users. "Submit" files can be run by just typing the file name, without the submit command, so you can run them just like compiled programs. Submit files can also call other submit files. (See COMMAND FILES in csh(cmnd)).

As delivered, C/NIX waits after every 23 lines of screen output for you to type any key. This is called "paging". If you don't like it, you can turn it off for everything but the **help** command; see set(cmnd) or patches(info).

Many of the built-in commands have been enhanced to allow a list of arguments, instead of just a single file specification. For instance the ERA command (also called **rm** — see rm(cmnd)) may be given more than one pathname or "wildcard" pathname ("ambiguous filename" in CP/M parlance — see wildcard(info)) for deletion in a single request.

A simple on-line documentation mechanism, the **help** command, has been built into the C/NIX shell (command processor), providing pages from this manual on demand. (See help(cmnd).)

If you are in a subdirectory, you still have access automatically to commands which are in the top level directory of the current disk, disk A, or the disk from which C/NIX was initially run. So you will want to put all your frequently used command files in, probably, the top level directory of disk A.

Also, if a running program looks for a disk file which is not in the current directory, it will find the file of the same name in any of these top level directories, if the SYS attribute of the file is set (see chmod(cmnd)). So you can keep library files and other common non-command files in one place too.

SEE ALSO

For more information on the features of C/NIX, you can browse through the other pages of this manual. In particular, see help(cmnd), commands(cmnd), csh(cmnd), ioredir(info), pathname(info), pipes(info), walk(cmnd), wildcard(info).

NAME

ioredir — Input/Output Redirection and Logging

SYNOPSIS

```
command param1 param2 ... > outfile
command param1 param2 ... >> outfile
command param1 param2 ... < infile
command param1 param2 ... >+ logfile
command param1 param2 ... <+ infile
command param1 param2 ... >& out_and_errorfile
command param1 param2 ... < ( explicit input ... )
command param1 param2 ... [ >>+ | >>& | >+& | >>+& ] out_log_err
command param1 param2 ... [ >! | >!+ | >!& | >!+& ] out_log_err
```

DESCRIPTION

C/NIX lets you redirect the console input or output of a command. That is, what would normally appear on the screen can be saved on a file instead, or in addition. Also, instead of taking typed input from the keyboard, the program can be made to take it from a file.

C/NIX also lets you save the keyboard input on the output file. Finally, it allows C/NIX error messages (output using the BDOS "direct console output" primitive) to be redirected with the normal output.

The first SYNOPSIS form (>) is the simple case of output redirection. In this case, output which would have appeared on the terminal is instead saved in the specified outfile. Outfile must be the name of an output device, or a new disk file; the shell will complain if it is an existing disk file (except see ! below).

Legal output devices are as follows:

```
lst:  The listing device,
con:  The console (ensures output goes to console
        even if some enclosing redirection is in force;
        bypasses console output paging if in effect),
pun:  The punch device,
err:  The "error" device, namely Direct Console Output,
        bypassing BDOS processing,
nul:  The Bit Bucket (output discarded).
```

The second SYNOPSIS form uses >> which means add the output to the end of the outfile if it already exists. If the file does not exist, it will be created.

The third SYNOPSIS form (<) is simple input redirection. Instead of reading from the terminal, the program takes its input from the specified infile when it requests "console input."

The fourth SYNOPSIS form (>+) is used when a permanent record ("log") of a program execution is desired in a logfile, including both the program's console output and any console input. When input is on the terminal, any logged program output will also appear on the terminal.

The fifth SYNOPSIS form (<+) cause console input to be redirected from a file, but with the side effect that it is echoed on the terminal as it is read. This is useful during debugging of a program when the input is stored in a file.

The sixth SYNOPSIS form (>&) causes redirection of both normal console output and "error" output (defined to be anything output using the BDOS direct console

output primitive, system call 6).

The seventh SYNOPSIS form (< (...)) is useful primarily within command files. The text within parentheses is provided when the command requests console input. The text may be several lines, and may include nested balanced or quoted parentheses. The final parenthesis becomes a carriage return, followed by an end-of-file (control/Z) when read by the command. Note that this replaces the XSUB feature of standard CP/M (XSUB itself is not supported).

The eighth SYNOPSIS form lists additional legal combinations of input logging, output redirection, appending, and error redirection. In all these cases, the + means input logging, & means error redirection, and >> means appending.

The ! flag in the final SYNOPSIS form allows overwriting of existing files by redirection. Without the ! flag, a file specified for output redirection/logging with a single > must NOT already exist.

If the input or output is to be redirected from or to the same file for a sequence of commands, with the later ones picking up where the former ones leave off, then the commands can be grouped into a command file (see csh(cmnd)) or directly with parentheses (see example below).

Although not illustrated in the SYNOPSIS, both input and output may be redirected for a single command.

EXAMPLES

```
ddt prog.com <+ debug.txt >+ progddt.log
```

This example will start DDT with the given program, taking commands from the file debug.txt, showing them on the terminal as they are read (due to <+), and logging both input and output in the file progddt.log (due to >+).

```
(type cover.txt; nroff body.nr
   type backcovr.txt) > lst:
```

This example runs a series of commands, and sends all of their output to the listing device.

NOTES

Enclosing commands in parentheses is equivalent to creating a command file with the enclosed text and then running it. The text may be several lines, and the end of a line (unless quoted) is equivalent to a semi-colon.

The concept of "error" output is not defined in standard CP/M, so C/NIX has introduced the convention that "direct" console output should be considered "error" output.

The actual physical devices associated with the output devices lst:, con:, and pun: may be adjusted using the normal CP/M STAT command, such as stat pun:=up2:.

Even if paging of output to the console is selected (set(cmnd)), the con: device will bypass the paging.

SEE ALSO

csh(cmnd), intro(info), pipes(info), quoting(info)

LIMITATIONS

In some cases, output may appear double spaced when input is redirected. This occurs when the program thinks it is reading the keyboard, where lines are terminated with the return (CR) character, rather than a file, which uses a CF-LF pair. The program will echo the CR as CR-LF, and the LF as LF, giving two LFs (line feeds) at the end of each line. Programs compiled using C/80 exhibit this problem, but since C/80 contains its own I/O redirection, the problem can be avoided by using \< instead of < to bypass the shell and use C/80's redirection instead.

The amount of text which may be provided using the parenthesis feature (< (...)) is severely limited. (All commands are limited to about 250 characters total.) This limitation can be overcome by creating a separate file with the input for the command.

NAME

patches — Patches to the C/NIX System

SYNOPSIS

Address (hex)	Meaning	Default	Legal Values
022A	First temporary drive letter	'J' = 4A	'C'...'M' = 43...4D
022C	Last temporary drive letter	'M' = 4D	'F'...'P' =46...50
0254	Drive for pipes (0 = use C/NIX drive)	0	'A'...'P' =41...50
0255	Commands drive (0 = not used) (searched for commands instead of drive A)	0	'B'...'P' =42...50
0256	Help drive (for help dir.) (0 = use C/NIX drive)	0	'A'...'P' =41...50
0257	Confirm required if overwriting files in cp, mv	1	0 = no, 1 = yes
0258	Verbose flag: echo commands before execution in .sub file	0	0 = quiet, 1 = echo
0259	SYS-bit required for universal visibility of commands in top-level directory.	0	1 = req'd, 0 = not
025C	Page screen output.	1	1 = yes, 0 = no
025D	If paging, number of lines/page	23 dec.	15...127 dec.
025E	ASCII code to warn user at end of page (e.g., 07 = bell, 3F = '?', 0 = none, etc.)	bell	Any ASCII char.
0260	Maximum user number to use for creating subdirectories.	1F	0F or 1F

DESCRIPTION

Certain aspects of the C/NIX system may be altered by patching the file **cnixhigh.sys**. The following example shows how to do this. It changes three of the drives above from their old defaults.

IMPORTANT: Since the SAVE command does not work under C/NIX, it is necessary to perform this procedure under CP/M, not C/NIX.

```
A>ddt b:cnixhigh.sys     — Load cnixhigh
NEXT  PC
0880 0100                — First free byte, and start
-s0254                   — Replace the pipe disk
0254 00   43             — 'C'
0255 00   44             — Make 'D' the commands drive
0256 00   .
-s22A                    — Replace the first temp drive
022A 4A   47             — 'G'
022B 00   .
-g0
A>save 8 b:cnixhigh.sys — Save (8 is (0880-0100) / 100 hex)
```

Now test it as follows

```
A>b:
B>cnix                    — Load C/NIX with patched cnixhigh
 ...
B$ echo help/foobar       — Test minimum temp disk letter
G:foobar

B$ ls -l /c | cat         — Test that pipes end up on /C
...
-rw-  0.0k  pipe0a.$$$    — Sure enough!
...

B$ ls -l /b | cat         — Test that they are NOT on /B
...
                    — Check that commands on D are found
B$ cp grep.pre /d/newgrep.pre        — Copy "grep" to D
B$ /d/newgrep "NOTES" help/intro.hlp — Should work
B$ newgrep "NOTES" help/intro.hlp    — Should also work
```

NOTES

The set command allows some of these patch locations to be changed in memory. Modes that can be changed are verbose, confirmation, and paging. The changes last only as long as C/NIX is not exited or rebooted.

At least four temporary drive letters should be provided. These are used as logical disks by the shell (see pathname(info)).

When a commands drive is specified, it is searched after the current drive, and before the C/NIX drive, instead of drive A. Only the top-level directory on the commands drive is searched.

SEE ALSO

csh(cmnd), cp(cmnd), help(cmnd), mv(cmnd), pathname(info), set(cmnd)

NAME

pathname — Hierarchical Directory Pathname Syntax

SYNOPSIS

```
/x/dir1/dir2/ ... /name.ext
../dir1/dir2/ ... /name.ext
x:dir1/dir2/ ... /name.ext
dir1/dir2/ ... /name.ext
```

DESCRIPTION

C/NIX supports a tree-like hierarchical directory structure. Each disk may have sub-directories along with files, and each sub-directory may have further sub-directories and files. This feature allows each directory to remain smaller, containing a logically related set of programs and data.

Files and directories in various directories are referred to by pathnames. The top-level directory of each disk has a pathname of the form /x where x is the disk letter. Each file or sub-directory in the top-level directory has the pathname /x/ followed by the file or directory name. For example, if the top level directory of disk a contains a directory help, the pathname of that directory is /a/help.

Once you have constructed the pathname of any directory, the pathname of any file or sub-directory within it is the pathname of the directory, followed by a slash (/), and then the name of the file or sub-directory. So the file mv.tin in the help directory of the preceding example has the complete pathname /a/help/mv.tin.

CP/M already has the concept of the current disk or logged-in disk. The letter of the current disk appears in the command prompt: for example, A> on CP/M, or A$ on C/NIX. If you give a file name without a disk letter (that is, file as opposed to b:file), the current disk is assumed.

In C/NIX, there is also a current directory on each disk. The current directory is determined by the most recent chdir command on that disk. The current directory on the current disk is the current working directory. The full pathname of the current working directory may be seen with the pwd command (see chdir(cmnd)).

If you give a file name without indicating a disk or directory, the file is assumed to be in the current working directory. (You can also give a pathname starting with a directory name, as in the fourth SYNOPSIS form; the path will start from the current working directory.)

For convenience, the parent directory of a sub-directory may be referred to by the special name .., allowing pathnames to specify traversing both "up" and "down" the hierarchy. This is illustrated with the second form under the SYNOPSIS. A single dot . represents the current directory itself, and is particularly useful for the cp(cmnd) and mv(cmnd) commands when copying/moving a group files into the current directory.

The walk command prints out all the sub-directories of the current directory. It can also execute a command in each sub-directory, allowing an operation to be performed on files in various directories.

NOTES

A sub-directory is represented as a file in its enclosing, or parent, directory, with a filename extension of .sd? where ? is a letter from 1 to 5 or a to z. A total of 31 sub-directories may be created on a single disk (see mkdir(cmnd)). When referring to a sub-directory, only the name part of the filename is used. (I.e. never mention the .sd? extension-part).

Files in the top-level directory of a disk which have the SYS flag set (see chmod(cmnd)) are accessible from any sub-directory. This works by having BDOS open (function 15) automatically check the top-level directory (i.e. user 0) whenever a requested file is not found in the current directory. File attribute f8 is set in the FCB so that subsequent reads and writes will reference the correct file. This is particularly useful for executable programs and program overlays, which are frequently referenced when working in a sub-directory.

The C/NIX shell recognizes the hierarchical directory pathname syntax, and converts it to a form acceptable to CP/M programs, by temporarily defining "logical" disks with letters from j to m (see patches(info)), and replacing the hierarchical pathname part with simply j:, k:, etc. Any parameter which contains un-quoted slashes (/) is translated in this way by the shell (see quoting(info)).

SEE ALSO

chdir(cmnd), chmod(cmnd), cp(cmnd), mkdir(cmnd), mv(cmnd), patches(info), quoting(info), walk(cmnd)

LIMITATIONS

For a single command, only four temporary logical disks may be defined (j, k, l, and m), thus limiting the number of sub-directories which can be referred to with a single set of parameters. This may be overcome by defining current directories of interest on the various disks with chdir, and using the x: notation, or by patching cnixhigh.sys to allow more temporary disks (see patches(info)).

Explicit use of the "user" feature of CP/M is not supported. There is no "user" command, and use of [Gnn] in PIP or the "setuser" BDOS system call is ignored.

NAME

pipes — C/NIX Pipes

SYNOPSIS

command1 param1 ... | command2 param2 ... | command3 ...

DESCRIPTION

Pipes allow several programs to be run in succession, with the output from one program being passed to the next program as its input.

Pipes are convenient when the output of a command should be sorted or reformatted in some way, before being saved permanently in a file. Programs designed to do this kind of sorting or reformatting are called filters.

C/NIX uses the vertical bar | to separate commands in a pipe.

EXAMPLE

asm foobar | grep "error"

This example assembles foobar, and then passes the output through the filter grep which outputs only the lines of its input which contain the string 'error' somewhere within them.

NOTES

Because CP/M is not a multi-tasking system, only one of the commands actually runs at a time. All of its "console" output is collected in a temporary pipe file, and then the next command in sequence is run, with its console input redirected to come from this temporary file.

Normally, the pipe files are created on the C/NIX disk, but C/NIX can be patched to use another disk; see patches(info). Because all of the output of one command must be collected before the next can be run, the pipe file disk must be writable, and have enough room to hold the data.

The C/NIX shell names the temporary pipe files /x/pipe0?.$$$ where x is the pipes disk, and ? is a character from 'a' to 'z'. The files are automatically deleted after use.

SEE ALSO

ioredir(info), patches(info), grep(cmnd)

NAME

quoting — Quoting and Escape Characters

SYNOPSIS

```
\x
' ... '
" ... "
( ... )
```

DESCRIPTION

Various characters have special significance in C/NIX commands. In particular, at various times, the following characters are interpreted specially:

? * / ; ( ) < > | $ \ ' " <SPACE> <RETURN>

If you want to use one of these characters in a command, you may need to "slip it past" the C/NIX shell to let the program "see" it. To do this, you must quote the character. You can quote a single character by prefixing it with the backslash (\) character. Alternatively, a string of characters may be quoted with matching apostrophes or double quotes.

A common problem occurs with CP/M commands which take "switches" containing the / character on the command line. The error message "Bad directory" may be given. To avoid this, enclose the entire argument in quotes.

Within quotes or parentheses, some but not all of the above characters lose their significance. In particular, within single quotes (apostrophes), all but backslash, <RETURN>, and single quote itself lose their significance. Within double quotes, <RETURN>, dollar sign, backslash, and double quote itself are still significant.

Parentheses defer the processing of all but dollar sign. Thus commands within command files may be given (multi-line) input which depends on the parameters to the command files.

EXAMPLES

```
180 foo,clibrary,foo\/n\/e
                    — / characters must be quoted.

ddt < (s$1
20
21                  — execute ddt, substitute at address (s$1)
22                  —  specified as first parameter to command
.                   —  file.
g0)

grep "and/or " thesis.txt
                    — Slash and space must be quoted.
```

NOTES

The amount of text which may be included within quotes is limited to a single line. The amount of text which may be enclosed within matching parentheses is limited to about 250 characters.

SEE ALSO

ioredir(info)

NAME

trouble — Trouble Shooting on C/NIX

SYNOPSIS

Problem	Possible Causes
A command containing special characters, such as ( ) / or \, does not work or gives error messages.	These characters have special meaning to C/NIX. See quoting(info).
The shell immediately exits after it is invoked with cnix.	You did not select the disk on which cnix.com resides before invoking it.
C/NIX exits instead of returning with a shell prompt.	You removed the C/NIX disk from the drive it started on.
The shell is slow in prompting after a program finishes.	Any non-relocatable program is loaded over the shell, and so the shell must be re-loaded after it completes. This process is fastest if cnix.com is the first file copied onto the diskette so it is close to the directory tracks, and contiguous. Also, some CP/M configuration programs allow the user to reduce the "stepping" time. Modern drives can handle a 6 millisecond step rate.
Files seem to be missing from the directory.	Files with the SYS flag set are not listed in the short form of ls or dir. Files in other directories are not listed unless that directory is also specified. The overlays for an editor or compiler are frequently required to be in the current directory. C/NIX will find files in the top directory if the SYS flag is set (chmod +s).
A file cannot be moved or removed.	Files with the "write" flag off cannot be moved or removed Use chmod +w to set the write flag. The disk is nearly full. Move consists of a copy followed by a delete, requiring room for two copies of the file temporarily.
Pipes don't work, or the disk fills up during a piped command.	The entire output of a piped command is collected in a temporary pipe file. The pipe file disk must be writeable, and with enough free space to hold it.
The output of a program cannot be redirected with >.	The program uses direct console output (try >&), or goes directly to the BIOS part of the operating system or to the hardware (hopeless).

Redirected program output is double spaced (extra blank lines inserted).	This happens with some programs written in C/80, because they add an extra line feed when writing to what they think is the screen. Redirect with \> instead which uses C/80's I/O redirection.
The input of a program cannot be redirected with <.	Same causes as above.
Tabs don't line up properly on 8-column intervals	During input, when output is redirected, tabs may echo improperly, due to an obscure bug in CP/M. The tabs will expand properly when viewed later. Try the < (...) feature instead (see ioredir(info)).
The SAVE command doesn't work.	SAVE is not supported by C/NIX. Patching must be done under CP/M.
The XSUB command doesn't work.	XSUB is not supported by C/NIX. Use input redirection instead (ioredir(info)).
A spooler (like DESPOOL) or other "background" or resident program does not work.	These programs "poke around" in CP/M, and not all can work with C/NIX. Those that can (like DESPOOL) must be loaded BEFORE C/NIX is run.
Hard disk backup procedures do not save all directories.	C/NIX uses user numbers 1 to 31. Some backup procedures may not save user numbers over 15 (directories .sdk and above). Use cp to copy files to disk instead, or patch C/NIX not to use user numbers over 15 (patches(info)).
Programs intended for a specific CP/M implementation will not run.	Certain programs (e.g., versions of Microsoft BASIC for some Heath/Zenith machines) refuse to run on other operating systems. These programs may not recognize C/NIX as a legal system. Exit from C/NIX to run the program.

DESCRIPTION

This section attempts to suggest probable causes for anticipated problems while running C/NIX. If after considering this list, you still cannot solve the problem, see the trouble reporting procedure at the end of the Introduction to the printed C/NIX manual.

SEE ALSO

patches(info)

LIMITATIONS

Some of the limitations implied in the probable causes above should be removed from C/NIX or CP/M.

NAME

wildcard — Filename Wildcards '?' and '*'

SYNOPSIS

? matches any single character of a filename.

* matches any number of characters at the end of the "name" part or the "extension" part of a filename.

DESCRIPTION

Wildcards are characters (? and *) used in filenames in commands in order to refer to a set of files with similar names, instead of a single file. C/NIX supports almost the same wildcard conventions as CP/M.

CP/M documentation refers to a filename pattern containing ? or * as an ambiguous file name(afn). In C/NIX, ? and * are called wildcards, and the filename is wildcarded.

As in CP/M, a ? takes the place of any character within the 8-character name-part or 3-character extension-part of a filename. In addition, a * may be used at the end of the name-part or extension-part, and it is equivalent to a string of question marks.

If a * appears at the end of a name, and no extension is specified, all extensions are matched. This is compatible with UNIX, but differs from CP/M, in which only the null extension would be matched.

EXAMPLES

a*.c matches all files whose name starts with a, and whose extension is .c.

cnix?b.a* matches all files whose name matches cnix?b and extension starts with a.

foo* matches all files whose name starts with foo, regardless of extension. The CP/M equivalent would be foo*.* (which also works on C/NIX). Use foo*. to match files with no extension.

NOTES

At the end of the name-part or extension-part, ? matches a blank (this is consistent with CP/M).

Unlike in UNIX, to maintain compatibility with existing CP/M programs, wildcarded filenames are expanded ONLY for the following commands:

chmod(cmnd), cp(cmnd), dir/ls(cmnd), era/rm(cmnd), grep(cmnd), mv(cmnd)

In addition, existing CP/M programs, or your own programs, may recognize the wildcard characters. A _main startup routine is provided on the C/NIX (source) disk for this purpose (see main(subr)).

SEE ALSO

intro(info), pathname(info), ls(cmnd), main(subr)

LIMITATIONS

The limitation that * may only appear at the end of the name-part or the extension-part should be removed (from CP/M as well).

NAME

commands — Summary of C/NIX Commands

SYNOPSIS

```
bye                                 — Leave the C/NIX shell (exit)
cat file1 file2 ... > outfile       — Concatenate text files
cat < ( ... )                       — Output text in parens
chdir                               — Change to top-level directory
chdir dir                           — Change to a new directory
chmod [+w|-w|+s|-s] file1 ...       — Change "mode" of files
cd                                  — Synonym for chdir
cp filefrom fileto                  — Copy a file
cp [-f|-c|-v|-q] file1 ... dir      — Copy files to new directory
csh [-v|-q] cmndfile param1 ...     — Invoke sub-shell on command file
dir [-lfdt] pattern1 pat2 ...       — List directories (ls)
dir [-lfdt]                         — List current directory (ls)
echo param1 param2 ...              — Echo parameters to console (csh)
era [-f] file1 file2 ...            — Erase files (rm)
exit                                — Exit the C/NIX shell
grep "pattern" file1 ...            — Search files for a pattern
grep "pattern"                      — Search console input for a pattern
help topic1 topic2 ...              — Display help information
help                                — Display list of help topics
ls [-lfdt] pattern1 pat2 ...        — List directories
ls [-lfdt]                          — List current directory
man topic1 topic2 ...               — Display pages from manual (help)
man                                 — Display list of manual pages (help)
mkdir dir1 dir2 ...                 — Make directories
mkrel file100 file200 file.pre      — Make page-relocatable program
mv [-f|-c] oldname newname          — Move/rename a file
mv [-f|-c|-v|-q] file1 ... dir      — Move files to new directory
pwd                                 — Print pathname of working directory (chdir)
ren oldname newname                 — Rename a file (mv)
ren newname=oldname                 — Rename a file (mv)
rm [-f] file1 file2 ...             — Remove files
rmdir dir1 dir2 ...                 — Remove directories (mkdir)
set [+|- [vcbp?]] ...               — Set certain user interface parameters
submit cmndfile param1 ...          — Submit command file (csh)
type file1 file2 ...                — Type text files (cat)
walk [-b]                           — Walk directory tree
walk [-b] command param1 ...        — Walk and execute command
```

NOTES

Each of these commands is described within this (cmnd) chapter, generally on a page devoted to that command. In cases where the command is described elsewhere, the name of that manual page is given in parentheses.

Optional flags are given in brackets, with alternatives separated with vertical bars. Ellipses (...) are used to represent a list of files, etc.

All of the above are recognized within the shell, except for grep and mkrel. All but these two and chmod, mkdir, and rmdir, are also implemented entirely within the shell. The commands chmod, mkdir, and rmdir are implemented by cnixutil.pre.

SEE ALSO

intro(info)

NAME

cat — Concatenate or Type Files

SYNOPSIS

```
cat file1 file2 ... > output
cat file1 >> file2
cat < (... predefined text ...)
type file1 file2 ...
```

DESCRIPTION

The cat command (alias type) can be used to concatenate ASCII files, type files, or display predefined text. It simply reads each file, and outputs it to the console. By using output redirection, the files can be effectively concatenated.

The second form shows the use of output appending (>>) to concatenate one ASCII file onto the end of another.

If no arguments are given to cat, it simply copies its console input to the console output. The third form above shows how this can be used in a command file to display some predefined text (or see also echo(cmnd)).

Type is a synonym for cat in C/NIX, and may be used in all of the same ways, as well as in the more conventional file display use illustrated in the last SYNOPSIS form.

The filename - represents by convention the console input. A user may thus insert some console-provided text between two files in a concatenation:

```
cat header.txt - trailer.txt > combo.txt
```

After copying header.txt, cat will wait for input from the console, copying it to console output until it receives the CP/M end-of-file character (control/Z). It will then copy over trailer.txt.

NOTES

In the first SYNOPSIS form above, the file output is created before the command actually begins, and if it is also mentioned as one of the input files, an infinite loop will be created.

Cat looks for the CP/M ASCII end-of-file indicator (control/Z), and hence cannot be used to concatenate binary files.

SEE ALSO

cp(cmnd), ioredir(info), echo(cmnd)

LIMITATIONS

Cat does not properly deal with wildcarded filenames. Only the first file matching the pattern is displayed. This limitation exists because, when concatenating, the order presumably matters, and at the moment, the shell does not guarantee any specific order for files which all match the same wildcarded pattern. Alphabetical order would be consistent with UNIX, and could be convenient.

NAME

chdir — Changing/Printing Current Working Directory

SYNOPSIS

```
chdir dir
chdir
cd dir
cd
pwd
```

DESCRIPTION

The current working directory represents the current focus of activity on C/NIX. It is like the current logged in disk on CP/M. Filenames without a disk letter prefix (no x:), refer to files within this directory. Pathnames which do not start with a slash (/) are relative to this directory (relative pathnames). By default, the ls command lists the files and sub-directories within the current working directory. A pathname of simply . may be used to refer to the current working directory (or ./; see LIMITATIONS below).

The chdir command (alias cd) changes the current working directory to be some new directory. With no parameter, the top-level directory of the current disk is selected. Otherwise, the specified directory is selected. The disk of this new directory becomes the current disk, and its letter is displayed as part of the C/NIX prompt.

After selecting a particular directory on a disk as the current directory, it continues to be accessible using simply the disk letter prefix (x:) instead of its full pathname, until a new directory is selected to be current for THAT disk. Thus at any one time, C/NIX keeps track of a current directory for every disk. The one for the current disk is considered the current working directory.

The pwd command displays the full pathname of the current working directory, and is handy to answer the question "Where am I!?"

LIMITATIONS

The only C/NIX commands that know about . and .. are ones such as rm(cmnd) and cp(cmnd) which deal with directories. For other commands (e.g., echo(cmnd)) and non-C/NIX programs which want a drive name as argument (e.g., A:), it may be necessary to use ./ and ../ as pathnames, in order to force the shell to provide the correct equivalent. The shell recognizes only pathnames containing the character /.

SEE ALSO

csh(cmnd), mkdir(cmnd), pathname(info)

NAME

chmod — Change File Mode Flags

SYNOPSIS

```
chmod +w file1 file2 ...
chmod -w file1 file2 ...
chmod +s file1 file2 ...
chmod -s file1 file2 ...
chmod [ +ws | -ws | +s -w | +w -s | ... ] file1 file2 ...
```

DESCRIPTION

The **chmod** command allows the user to change the file mode flags of a file. Each C/NIX file has two mode flags, a writeable flag, and a SYS flag.

When files are created, they are by default writeable. However, this flag may be cleared, after which the file cannot be moved or removed. **Chmod +w ...** sets the writeable flag, while **chmod -w ...** clears it.

The SYS flag controls whether files are visible in short form directory listings. By default the flag is off, and the file appears in the listing. However, when the SYS flag is set (with **chmod +s ...**), the file is invisible, and no longer appears in the short form listing. In the long form listing (**ls -l**), the SYS flag prints as an **s**. See ls(cmnd).

Both the "writeable" and SYS flags may be turned on or off in a single **chmod** command, as illustrated in the last form under SYNOPSIS above.

NOTES

These flags correspond to CP/M flags. The C/NIX "writeable" flag is the complement of the CP/M "read-only" flag. The C/NIX SYS flag is the same as the CP/M SYS flag. The flags are implemented using the high order bit of the first two characters of the filename extension, when stored in the directory.

These flags have less effect on directory files (e.g., **help.sda**). If the directory file is not writeable, then the directory cannot be removed, but components can still be added to or removed from it. The SYS flag, if set for a directory file, will cause the directory file to be omitted from a short form listing of its parent directory, but a short form listing of the directory itself will be unaffected.

Commands in the top-level directory of a drive are normally accessible from any sub-directory. If the SYS flag of a file in the top-level directory is set, the file is also accessible to be opened by any running program. This allows library files, for example, to be stored in the top directory for access from any directory. The requirement of the SYS flag being set can be eliminated by patching **cnixhigh.sys** (see patches(info)).

SEE ALSO

cp(cmnd), ls(cmnd), pathname(cmnd)

LIMITATIONS

Clearing the "writeable" flag on a directory file should perhaps prevent adding or removing components from the directory.

NAME

cp — Copy a File or Files

SYNOPSIS

```
cp [ -f | -c | -v | -q ] filefrom fileto
cp [ -f | -c | -v | -q ] file1 file2 file3 ... dir
```

DESCRIPTION

The cp command copies one file to a new one, or copies a set of files to another directory. Cp will ask for confirmation if a file already exists with the new name. (-f flag "forces" copy without asking for confirmation, regardless of whether the target file exists.) The rm command may be used to remove existing files before copying to them.

The first form of cp takes two filenames; it copies the first to the second. This is equivalent to a simple use of the PIP command.

The second form of cp takes one or more filenames, and a directory name. All the files are copied onto files with the corresponding names in the specified directory. The filenames may contain wildcards, in which case all matching files will be copied into the specified directory. The directory must exist, but if it contains any files with the same names as those being copied, confirmation will be requested.

The -f flag "forces" copy without confirmation. This can be made the default by patching cnixhigh.sys (see patches(info)), in which case -c overrides the default and requests confirmation again.

The -v flag (for "verbose") causes each file name to be echoed as it is copied. This may be made the default by patching cnixhigh.sys (see patches(info)), in which case -q overrides the default and requests "quiet" mode again.

The source files are unaltered. To rename a file, or move it to another directory, see mv(cmnd).

SEE ALSO

mv(cmnd), rm(cmnd), patches(info), pathname(info), wildcard(info)

NAME

csh — The C/NIX(tm) Shell — Command Formats

SYNOPSIS

```
B>cnix
csh
csh [-v|-q] cmndfile param1 param2 param3 ...
cmndfile param1 param2 param3 ...
(command1 param11 param12 ...; command2 param21 ... )
```

DESCRIPTION

The C/NIX shell is the program which displays the command prompt (for example, A$), reads the commands you type, and executes them. It also executes commands files (also called batch or submit files). (The CP/M equivalent is called the CCP.)

Our discussion of the shell breaks down into two topics: how to invoke the shell, and what features and commands the shell provides. Normally, you won't need to invoke the shell at all, since it comes up automatically when C/NIX is entered, or when you run a command file. So first time users may want to skip to the next heading now.

The C/NIX shell may be initiated in several different ways, as illustrated by the various forms under SYNOPSIS above.

The first SYNOPSIS form shows how the shell is first invoked as part of C/NIX initialization. The disk on which **cnix.com**, **cnixhigh.sys**, **cnixutil.pre**, and **help.sda** all reside MUST be the current disk when **cnix** is first invoked, or else it will immediately exit.

The second form (**csh**) invokes a sub-shell. In effect, this places the current shell aside and drops down into a new one. Exiting the new sub-shell (see exit(cmnd)) returns to the old shell. This is useful for wandering off temporarily to various other directories, since exiting returns to the original directory. This form is also useful when a list of shell commands are constructed by some program, and then piped into the shell (see pipes(info)).

In the third form, the shell is run with the name of a command (**.sub**) file from which it is to take commands to be executed. The remaining parameters in the command replace occurrences of $1, $2, etc., in the command file. The optional **-v** flag ("verbose" or "verify") causes commands from **cmndfile** to be echoed before execution. This can be made the default by patching **cnixhigh.sys** (see patches(info)), in which case the **-q** flag may be given to request "quiet" mode again. See "Command Files" below for more details.

Command files can also be executed simply by typing the name of the command file and any arguments, as shown in the fourth form (**cmndfile param1 ...**). In this case, the shell is invoked implicitly. This form will search multiple directories for **cmndfile** (see below).

The final form ((**command1 ...**)) shows the grouping of commands, useful for piping or redirecting their I/O as a whole. This is equivalent to creating a command file with the parenthesized text, and then running it. Again, the invocation of a sub-shell to process the commands is left implicit.

COMMAND FORMAT AND SEARCH PATH

The basic format of commands to the C/NIX shell is similar to CP/M:

```
B$ command_name param1 param2 param3 ...
```

The prompt reminds the user of which is the current disk (B in this case). The chdir(cmnd) command may be used to change the current disk.

If the command_name does not include a filename extension, then the shell will try **.com** (normal CP/M programs), **.pre** (page-relocatable programs, loaded above C/NIX shell, built by mkrel(cmnd)), or **.sub** (C/NIX command files).

If the command_name is not found in the current working directory, the shell will look for it in certain other directories. These directories make up the search path. This is very useful, because you don't have to keep copies of all your commands in each directory. You can just keep one copy in a directory on the search path, and use it from any directory at all.

The search path consists of the current working directory, the top-level directory of the current drive, the top-level directory of drive A, and finally the top-level directory of the drive from which C/NIX itself was loaded. A "commands" drive may be substituted for A in this search path by patching **cnixhigh.sys** (see patches(info)).

If the "SYS-bit required" flag is set in **cnixhigh.sys** (see patches(info)) then only commands with the SYS bit set (see chmod(cmnd)) will be found in the top-level directory of the current drive. This provides compatibility with CP/M 3.0. (Note that when a program tries to open a file during execution, the top-level directory is also searched automatically, but the SYS bit is always required in that case.)

Certain command names are built into the C/NIX shell. These commands are either implemented within the shell itself (for example, **cp**, **mv**, **ls**), or are implemented by a special utility program called **cnixutil.pre** (currently only **chmod**, **mkdir**, and **rmdir**). These commands will start and finish more quickly because they are built in to the shell, or are in a known directory, and the shell does not have to be reloaded after they finish.

Bye and **exit** are two names for the built-in command which makes the shell finish execution, and return to CP/M if a "top-level" shell, or return to the invoking shell if a sub-shell. End-of-file on a command file will also cause the shell to exit.

Several C/NIX commands may be entered on the same line by separating them with a semi-colon (;). Alternatively, you may use several lines to type a single command by typing a back-slash (\) immediately before the <RETURN> key (see quoting(info)).

The shell also provides for input and output redirection and logging (see ioredir(info)), as well as the connection of two or more commands with C/NIX pipes (see pipes(info)).

COMMAND FILES

As mentioned above, the C/NIX shell supports command files. When the command file is run, each line from the file is read and executed by the shell, substituting the actual parameters for $1, $2, etc. This is just like the CP/M SUBMIT command, but you don't have to type the word SUBMIT, just the name of the command (or submit) file.

Unlike the CP/M SUBMIT command, command files in C/NIX don't display the commands on the screen. You can run the shell explicitly with the **-v** ("verbose") flag to make echoing happen:

```
csh -v cmndfile param1 ...
```

or patch **cnixhigh.sys** to make verbose the default mode (see patches(info)).

Unless redirected, commands within command files receive their console input and output from the same place as when the sub-shell was invoked. This allows command files to act as normal programs, interacting with the user at the terminal, or as a filter in a pipeline.

For convenience, when a command is to be run with predefined input, the text may be included as part of the command file by enclosing it in parentheses, as follows:

```
command param1 ... < ( ... predefined input ... )
```

The predefined input may include references to the command file parameters using **$1**, **$2**, as usual.

To display prompts and general commentary on the terminal while a command file is running, an **echo** command is provided which simply echoes its parameters (after doing **$1**, ... substitution):

```
echo Please wait while I crunch on $1 and $2 ...
```

When this command is encountered by the shell, it will display on the console output "Please wait while I crunch on foo.txt and bar.c ..." (for example), presumably informing the user of a coming pause in output.

This same effect can be accomplished using:

```
cat < (Please wait while I crunch on $1 and $2 ...)
```

See above and cat(cmnd).

To include comments which are not echoed when encountered in the command file, start the line with a semi-colon (;), or introduce them with a double-dash (—). Comments continue to the end of the line:

```
; This is a full-line comment
cc $1.o=$1.c    — This is a partial-line comment
```

Conditional or repetitive execution can be accomplished by piping the output of a program into the shell. For example, suppose you write a program called if which evaluates its first argument, and then outputs its second or third argument depending on whether the result is true or false. This could provide a primitive conditional execution facility as follows:

```
if ($1 = -help) (
        echo Usage: Funclist source.ada output.lst
) (
        grep -n "^function" $1 | sort > $2
) | csh
```

If the expression evaluates to true (i.e. the first command file parameter were **-help**), then the **echo** command is piped to the sub-shell. Otherwise, the **grep | sort** command is piped to the sub-shell.

Remember, that in such an example, a program like the hypothetical **if** above is not actually executing the commands, but rather piping the text to the shell to interpret.

EXIT STATUS

If you are writing a program, and want it to abort any command file it is part of, you can do so by exiting with a negative exit status. This is done by calling the **exit** function provided in the file **cmain.c** (see main(subr)), or passing the exit status directly to BDOS function 108 (ignored by normal CP/M 2.2), and then returning, or jumping to address zero (warm start). Codes -256..-129 are user-definable fatal error codes. Codes >= 0 indicate success. Codes -128..-2 are reserved for CP/M 3.0. Code -1 is used to retrieve the current exit status (e.g., **x = bdos(108, -1)**).

Upon return from a program, the C/NIX shell retrieves the value using BDOS function 108, and if it is negative, it ignores the rest of the current command file or typed command line. The exit status is initialized to a positive value so as to accommodate those commands which do not set its value at all.

NOTES

If your own program has the same name as a built in C/NIX shell command, you can get to it by prefixing its name with its explicit disk letter, or by explicitly specifying the **.com** extension, or by renaming it with **mv**.

SEE ALSO

intro(info), ioredir(info), patches(info), pathname(info), pipes(info), quoting(info), wildcard(info), cat(cmnd), exit(cmnd), grep(cmnd), mkrel(cmnd)

LIMITATIONS

In a few cases, output files may appear double spaced when input is redirected; see ioredir(info) for details.

The number of parameters and amount of text forming a single command "line" are limited to about 30 parameters, and 250 characters of text. By quoting or parenthesizing, a single command "line" may in fact cross multiple lines. Nevertheless, these overall limits still apply.

Direct support for conditional and repetitive execution should be added to the shell (although see **if** example above).

NAME

echo — Echo a string — Display a String on the Console.

SYNOPSIS

echo [anything at all]

DESCRIPTION

The "echo" command displays the remainder of the command line on the terminal, or standard output. It can be used to display a message on the terminal during execution of a batch file.

NAME

exit — Exit C/NIX Shell — Return to CP/M or Invoking Shell.

SYNOPSIS

exit
bye

DESCRIPTION

The exit command leaves the C/NIX shell, returning to CP/M if this is a "top-level shell," or to the invoking C/NIX shell if this is a "sub-shell."

After exiting, the user is returned to the directory that was current at the time the shell was invoked.

NOTES

The bye command is synonymous with exit under C/NIX. An end of file will also cause the shell to exit (i.e., the end of a command file, or control/Z from the console).

SEE ALSO

csh(cmnd)

NAME

grep — Generalized Regular Expression Parser

SYNOPSIS

```
grep [-nvc] 'pattern' file1 file2 ...
grep [-nvc] 'pattern'
```

Patterns have the general forms:

```
'abc...*...?...[j-m]...xyz'
'^abcde'
'abcde$'
```

DESCRIPTION

The grep command is based on an old favorite from UNIX systems. The grep command searches a list of files (or the console input) for lines which contain a text string, or which match a pattern. Normally, the matching lines from the file are displayed on the console, although there are other options.

A pattern can be simply a text string In this case, grep just outputs all lines containing that string.

A more complicated, and general, search can be done using a pattern which is a regular expression. This is something like using wildcards (* and ?) in file names (see wildcard(info)), but more complex. In a grep pattern:

?	Matches any single character
*	Matches zero or more arbitrary characters
[abc]	Matches any one of characters in brackets
[a-m]	Matches any character in the given range
[^qz]	Matches any character but those following ^
^	Matches beginning of line
$	Matches end of line

Any other character matches itself only.

The flags (n, v, or c) control the matching or printing process:

-n	Print line numbers in front of text of line
-v	Output only lines which do NOT match
-c	Upper/lower case COUNTS in matches; otherwise, case is ignored.

When the -c flag is set, the characters on the command line are all taken to be lower case. Characters which are to be upper case must be preceded by a backslash (\). For example:

```
grep -c '\the' paper.txt
```

will only display lines with exactly "The" somewhere within them.

NOTES

The C/NIX grep command accepts a slightly different regular expression syntax than that on UNIX systems (in particular, ? instead of ., * instead of .*). This is more consistent with C/NIX filename wildcards.

Grep is not a built-in shell command. The program grep.pre may be placed in any directory normally searched.

LIMITATIONS

Having to backslash upper case for the -c flag is a bit baroque. Unfortunately, to be compatible with CP/M, upper/lower case distinctions must be ignored on command lines.

NAME

help — Help From the On-Line User's Manual

SYNOPSIS

```
help
help topic1 topic2 ...
man
man topic1 topic2 ...
```

DESCRIPTION

The **help** command (alias **man** for "manual pages") displays information drawn from the C/NIX User's Manual. The available information is organized into topics, or equivalently pages, just like the printed manual.

Help with no arguments shows the list of topics. **Help** with a list of topics displays the information for each topic on the console, one after the other.

The information is displayed 23 lines at a time. After displaying 23 lines, the computer waits for any key to be typed before continuing.

Each manual page is organized into the following sections:

NAME	— Name and descriptive title of manual page
SYNOPSIS	— A short summary of how to use the feature
DESCRIPTION	— A discussion of the topic, command, or subroutine
EXAMPLES	— Examples as appropriate
NOTES	— Interesting side issues
SEE ALSO	— Other manual pages of interest
LIMITATIONS	— Existing limitations or possible enhancements

The SYNOPSIS section gives brief, possibly cryptic examples of how to use the command or feature. Throughout the manual page, but especially in this section, a special notation is used to describe variations and optional command fields.

Anything in square brackets, [like this], is optional and may be omitted. Example: **rm [-f] file** means you can type either **rm file** or **rm -f file**; the **-f** is optional.

The symbol **...** means any number of the preceding object may be used. Example: **mkdir file ...** means you can have any number of file names after **mkdir** (but at least one).

The symbol | means either the symbol on the left or the one on the right may be used. Example: **chmod [+w|-w|+s|-s] file ...]** means you can say **chmod +w file**, or **chmod -w file**, etc. Sometimes you can use more than one of the alternatives; sometimes only one makes sense.

Switches are special arguments to commands. Following the UNIX convention, switches are a minus sign followed by one or more letters. When several letters are shown, usually any one or more can appear in one or more switches. Example: **grep -nvc 'pattern'** means you can follow **grep** with **-n**, **-v**, **-c**, **-nv -c**, **-c -v**, etc.

NOTES

The display of help topics shows them with their filename extensions **.hlp**. There is no need to type this extension when requesting help on a topic.

In order to save space on the distribution disk, the help file for intro(info)

has been edited. The printed manual page contains more information.

You can add your own help files by making text files, giving them the appropriate name with the .hlp extension, and putting them in the help directory.

If space is a problem, you may place the help files on a disk other than the one containing C/NIX by patching **cnixhigh.sys** (see patches(info))).

SEE ALSO

intro(info), patches(info)

LIMITATIONS

The information could be more efficiently encoded on the disk, instead of one topic per file.

NAME

ls — List Directory

SYNOPSIS

ls [-l | -f | -d | -t] name1 pat2 name3 ...
dir [-l | -f | -d | -t] name1 name2 pat3 ...

DESCRIPTION

The ls command displays information on files and sub-directories within a directory, or files which match a wildcard pattern (see wildcard(info)).

The dir command is a synonym for ls; it is provided for compatibility with CP/M.

The ls command alone, with no arguments, lists the names of all files and sub-directories in the current working directory.

If the -l flag ("long" format) is given, then ls displays the mode and size of each file or sub-directory, and the total space used by all the listed files and sub-directories.

The -f flag lists only files, not subdirectories. The -d flag lists only sub-directories. Normally, both are listed. The -t flag lists only the total disk space. See examples below.

In the long format (-l) listing, ls displays the "mode" of each file or sub-directory. Modes are set by chmod(cmnd) (or the CP/M stat command), and are a set of four flags:

1) 'd' for sub-directories
   '-' for files

2) 'r' for all files and sub-directories (for UNIX compatibility; all files are "readable" under C/NIX)

3) 'w' for read/write files
   '-' for read/only files

4) 'x' for "executable" files (defined to be those with extension .com, .pre, or .sub).
   's' for "system" files (will not appear in short form ls or dir listing)
   '-' for all other files and sub-directories

For example, drw- is a typical directory mode, -r-x is the mode for a read/only executable file.

The size of a file is given in kilo-bytes. The size of a directory is defined to be the space (in kilobytes) occupied by all files within it, or any of its sub-directories.

The totals given at the end of the listing include the total space occupied by all those files and sub-directories listed, and the total for the entire disk.

EXAMPLES

ls *.c *.asm Display all files in current directory with extension .c or .asm.

ls -d /d e: Display all sub-directories in top-level directory of disk D, and all sub-directories in current directory of disk E.

ls -lf help Display in long form a list of all files in **help** sub-directory.

ls -t /a Display the total space in use on drive A.

ls -t *.bak Lists the space used by all **.bak** files.

ls -dt Lists the space used by all sub-directories of the current directory.

NOTES

The flags may be combined into a single parameter such as **-lf** or **-td** or given separately as **-l -f** or **-t -d**.

The **-l** and **-t** flag of **ls**, combined with chmod(cmnd), largely obviate the need for the CP/M STAT command.

SEE ALSO

chdir(cmnd), chmod(cmnd), pathname(info), wildcard(info), walk(cmnd)

NAME

mkdir — Make and Remove Directories

SYNOPSIS

mkdir dir1 dir2 ...
rmdir dir1 dir2 ...

DESCRIPTION

The **mkdir** command creates one or more sub-directories. The name of the directory must not have any dots (.) in it. If a pathname with slashes is given (/), then only the last directory in the path is created. All of the others must already exist.

The **rmdir** command removes one or more sub-directories. The directories must already be empty (see rm(cmnd)). If a pathname with slashes is given, then only the last directory in the path is removed.

Sub-directories are represented by a file in the parent directory with the name of the sub-directory and the extension **sd?**, where ? is a character in the range **1** to **5** or **a** to **z**. The files of the sub-directory are stored under a CP/M user number determined by this last character. **1** to **5** are user 1 to 5, **a** is user 6, **b** is user 7, etc. up to 31.

NOTES

A single **mkdir** command cannot create both a directory and a sub-directory within it. For example, this will NOT work:

mkdir sources sources/pascal — Won't work

This on the other hand WILL work:

mkdir sources; mkdir sources/pascal — Will work

This is because the shell translates all pathnames to simple filenames before a command is executed. In the first case the **sources** directory does not yet exist, so that **sources/pascal** cannot be translated to **J:pascal** (for example). The second case works, because the former **mkdir** finishes before the shell attempts to translate **sources/pascal**.

A maximum of 31 sub-directories may be created on a single disk. If any CP/M user numbers from 1 to 31 are already in use on the disk, they can not be used as directories, reducing the number available.

It is not meaningful to attempt to make or remove a top-level directory, like /d.

Mkdir and **rmdir** are implemented by the special relocatable program **cnixutil.pre** on the C/NIX disk.

SEE ALSO

rm(cmnd), pathname(info)

LIMITATIONS

The 31 sub-directory maximum could be a troublesome limit for some very large disks. Unfortunately, this limitation cannot be easily removed while remaining compatible with CP/M. (Maybe it's time to step up to that 32-bit supermicro?)

NAME

mkrel — Make Page-Relocatable Program

SYNOPSIS

mkrel base100.com base200.com progm.pre

DESCRIPTION

The **mkrel** command creates a version of a program called a page relocatable program, which can be loaded and executed without removing the C/NIX shell from memory. This makes the program run faster, and is useful for small, frequently used commands.

Mkrel compares two **.com** format files to form a page relocatable program with the extension **.pre**. The first file must begin at 100 (hex) and the second at 200 (hex). Such files are usually created by assembling the same program twice with different ORG statements at the beginning. A page relocatable program can be run by the C/NIX shell, and will load above the end of the shell, instead of replacing it.

NOTES

Because page relocatable programs are loaded above the shell, they have significantly less memory available.

Control returns more quickly to the shell after the execution of a page relocatable program because the shell need not be reloaded from disk.

The format produced by **mkrel** is based on the format produced by the PREL program delivered with Heath CP/M-80 systems. Programs built with **mkrel** have a length in their second and third bytes, the code starting at the 257th byte, and a map of relocation bits at the end. There is one relocation bit for every byte of code. If the bit is off, the corresponding byte is to be loaded unchanged. If the bit is on, the corresponding byte and its preceding one are adjusted by the address where loading started.

It is normal for programs linked with certain linkers (e.g., Microsoft's L80) to have random values in uninitialized data areas. This will cause **mkrel** to complain about bytes differing by more than 1 in those sections of the files. As long as this only occurs in uninitialized data areas, the resulting **.pre** file will still load and execute properly.

SEE ALSO

csh(cmnd)

LIMITATIONS

A more compact format would be possible.

NAME

mv — Move a File or Files

SYNOPSIS

```
mv [ -f | -c ] oldname newname
mv [ -f | -c | -v | -q ] file1 file2 file3 ... dir
ren oldname newname
ren newname=oldname
```

DESCRIPTION

The **mv** command moves (renames) a file to have a new name, or moves a set of files to another directory, keeping the same file names. Mv will ask for confirmation if an existing file would be replaced, unless the -f switch is used (see below).

The first form of **mv** takes two filenames, neither of which should contain wildcards. The first file is renamed to the second name.

The second form of **mv** takes one or more filenames, and a directory name. The filenames may contain wildcards, in which case all matching files will be moved into the specified directory. The directory must exist, but if it contains any files with the same names as those being moved, confirmation will be requested unless the -f flag is used.

The -f flag "forces" the move to a new name or directory without confirmation. This may be made the default by patching (see patches(info)), in which case, -c overrides the default and requests confirmation again.

The -v flag (for "verbose") causes each file name to be echoed as it is moved. This may be made the default by patching (see patches(info)), in which case -q overrides the default and requests "quiet" mode again.

The **ren** command is only legal if the file remains in the same directory. It is equivalent to the first form of **mv**.

NOTES

The **mv** command is equivalent to **cp** followed by **rm**, except when the source and destination are in the same directory, in which case it is a straight rename.

The second form of **ren** is provided for compatibility with CP/M.

SEE ALSO

cp(cmnd), rm(cmnd), patches(info), pathname(info), wildcard(info)

NAME

rm — Remove File or Files

SYNOPSIS

rm [-f] file1 file2 ...
era [-f] file1 file2 ...

DESCRIPTION

The rm command (and its synonym era) can be used to remove disk files. If the filenames contain wildcard characters (i.e. ? or * — see wildcard(info)), then rm will by default ask for a confirmation. The -f flag will suppress this check, as will running the command non-interactively (e.g., from a command file).

When rm asks for a confirmation, the possible responses are:

n — Do NOT remove files matching the wildcarded pattern.
y — Remove files matching the pattern.
f — Remove the files, and suppress further confirmation checks for the duration of the command.

NOTES

The rm command should NOT be used to remove sub-directories. Use the rmdir command (mkdir(cmnd)) instead.

SEE ALSO

mkdir(cmnd), wildcard(info)

LIMITATIONS

It should probably be illegal to remove a directory file (like help.sda), because its associated sub-directory might not be empty. The error is not catastrophic, however, because if the file is simply recreated (e.g. echo > help.sda), the sub-directory full of files will become reaccessible.

NAME

set — Set Console Output Modes

SYNOPSIS

set [+cpbv?] [-cpbv?] ...

DESCRIPTION

The **set** command lets you turn off or on three modes which affect output on your screen. The modes are verbose, paging, and confirmation. The modes are turned on or off by the + or - flags, respectively. The letter(s) following the + or - determine which modes are changed.

The changes made by the **set** command will be in effect only until C/NIX is exited or the system is rebooted.

The **v** flag sets the verbose mode. If this mode is on, commands such as **cp**, **mv**, **rm**, and **set** itself tell you what they are doing as they do it. When verbose mode is off (the default), the commands just execute without any messages. In addition, with verbose mode off, command files (submit or shell files) do not echo the commands in the file; with verbose mode on, they do echo.

The **p** flag determines whether output to the console is paged or not. In paged mode, whenever 23 lines are output without any typed input, C/NIX pauses and waits for any character to be typed before proceeding. (This always happens in the **help** command, whether paged mode is on or off.)

In paged mode, a character can be output to alert you to the need to type a key to proceed. The **+b** switch will cause a bell to be output. The **+?** switch will cause a '?' to be output. Turning either of these switches off will cause no character to be output.

The **c** flag determines whether confirmation is requested by the **rm**, **mv** and **cp** commands when a problem is encountered. When confirmation mode is on (the default), these programs will not copy over an existing file without requesting confirmation. You can type four letters when confirmation is requested:

y - yes; remove the file.
n - no; don't remove the file.
f - fast; remove the file and stop asking.
x - exit; don't remove the file and quit now.

NOTES

All these modes can be set permanently by patching the system. See patches(info). If you want to change the modes from the default but don't want to patch the system, you can invoke the **set** command in **cnixinit.sub** (see intro(info)).

SEE ALSO

csh(cmnd), patches(info).

NAME

walk — Walk the Directory Tree

SYNOPSIS

```
walk [ -b ]
walk [ -b ] command arg1 ...
walk [ -b ] ( command arg ... ; command arg ... ; ... )
```

DESCRIPTION

The **walk** command walks the directory tree starting at the current working directory. That is, it goes to the current directory and all of its subdirectories, sub-subdirectories, etc. In each directory, it performs an action depending on the form of the walk command used. The order in which the directories are taken is either top-down (the default), or bottom-up (-b flag).

Walk with no arguments simply echoes the full pathnames of all directories.

If followed by a command or parenthesized command list, after echoing the directory pathname, **walk** executes the command (list). Any "wildcard" specifications are re-evaluated for each command (list) execution. However, pathnames with slashes (/) in them are always interpreted relative to the starting directory.

For example, to delete all files with the extension .bak in the current directory and all its subdirectories, use (cautiously) the command

```
walk rm -f *.bak
```

The default order of walking is top-down, which in a directory tree means the current working directory first, the the first sub-directory, then the first sub-directory of the first sub-directory, etc.

The -b flag requests that the walk be done bottom-up, which means going all the way out to a leaf sub-directory (one with no further sub-directories), and doing a directory only after all its sub(sub)directories.

NOTES

A simple way to determine the size of each directory on a drive is:

```
chdir /x; walk ls -ft
```

which will change to the top-level directory of the specified drive, and then display a file size total for each directory in the walk.

SEE ALSO

pathname(info), wildcard(info), chdir(cmnd), ls(cmnd)

NAME

subrs — C Subroutines for C/NIX

SYNOPSIS

bdos(code, arg)	— Call BDOS with code in C, arg in DE
fopen(fname, mode)	— Open a file, return a FILE pointer (fopen)
fclose(file)	— Close a given FILE pointer (fopen)
getc(file)	— Get a character given a FILE pointer (getc)
getchar()	— Get a character from the console (getc)
_main()	— Hidden main routine, expands wildcards, etc. (main)
malloc(size)	— Dynamically allocate memory chunk (malloc)
free(ptr)	— Free allocated memory chunk (malloc)
compress()	— Compress dynamic allocation "heap" (malloc)
putc(c, file)	— Put a character given a FILE pointer (putc)
putchar(c)	— Put a character on the console (putc)
fflush(file)	— Flush output for given FILE pointer (putc)
strany(c, str)	— Return non-zero if char. within str. (strutils)
strcmp(str1, str2)	— Return <0, =0, >0 after comparing strs. (strutils)
strcpy(to, from)	— Copy string, return ptr to end of "to" (strutils)
streql(str1, str2)	— Return non-zero if strs identical (strutils)
strlen(str)	— Return length of string (strutils)
mvbytes(from, to, num)	— Copy bytes (strutils)

DESCRIPTION

This (subr) chapter describes C subroutines written to work with C/NIX. Most of these routines will also work with normal CP/M systems.

> IMPORTANT CAUTION: The routines have only been tested with the C/80 2.0 compiler from Software Toolworks, Sherman Oaks, California. Some of them replace routines included with that compiler, in a way that is more directly compatible with the UNIX standard subroutine libraries. Equivalents for some of these routines are included with later C/80 versions.
>
> Using these routines with any particular C compiler, including C/80, may require replacing or removing parts of the I/O library provided with the compiler, or renaming functions in order to remove name conflicts. This is a job for an experienced programmer. Neither C/Craft nor The Software Toolworks can provide advice or assistance beyond the information in this chapter.

The routines are grouped onto manual "pages". For each routine above, the name of its manual page is given in parentheses after its description.

NOTES

These routines follow the C/80 machine language calling conventions, as follows. Arguments are pushed onto the stack as 16 bit values, leftmost argument first. If a function returns a value, it is in the HL register. The calling routine is responsible for popping arguments back off the stack. No registers are preserved.

NAME

bdos — C Interface to BDOS Calls

SYNOPSIS

```
x = bdos(code, arg);
```

DESCRIPTION

This subroutine loads **code** into the C register, **arg** into the DE register, and then calls BDOS (via low-memory jump vector at 5/6/7). The BDOS return value comes back as the function return value. This gives the C programmer direct access to all of the BDOS interfaces.

A complete set of C preprocessor definitions are provided in a file **bdos.h** on the C/NIX (source) distribution disk. The actual code is in **bdos.c**, written assuming the Software Toolworks C/80 calling conventions.

NOTES

When running with C/NIX, the jump vector at 5/6/7 has been altered to point to the **cnixhigh.sys** interface module. This is transparent to the programmer.

Bdos is provided for compatibility with C/80 2.0 and earlier. It is included with C/80 3.0 and later.

FILES

bdos.h, bdos.c

NAME

getc — Standard Buffered Character Input

SYNOPSIS

```
FILE *filep;
  ...
c = getc(filep);
```

```
struct gc_buf_rec getbuf;
getbuf.g_max = sizeof(getbuf.g_data);
set_gcbp(&getbuf);
  ...
c = getchar();
```

DESCRIPTION

These two get routines provide standardized buffered character input. Getc expects an opened **FILE** pointer (see fopen(subr)), or one of the two standard file pointers **stdin** or **stderr**. **Getc(stdin)** reads from the console using **getchar** (see below). **Getc(stderr)** reads from the console using the "direct console I/O" BDOS call (bypassing any C/NIX input re-direction).

Getchar reads from the normal console input, which may have been re-directed from a file by the C/NIX shell. If a previous call to **set_gcbp** has been done, then **getchar** uses the "read console buffer" BDOS call. Otherwise, it uses the single-character "console input" BDOS call. In either case, each call returns the next input character, returning the defined value **EOF** when end-of-file or control/z is reached (see the file **stdio.h**).

Set_gcbp sets up a console buffer for getchar. The buffer must be of the form defined in **stdio.h**, which is based on the console buffer required by the "read console buffer" BDOS call. The field **g_max** must be initialized to the length of the data area, as shown in the SYNOPSIS above.

NOTES

Unless in binary mode (see fopen(subr)), none of the get routines return the carriage return character (\r), but instead return a single C newline (\n) to represent end-of-line. For **getchar**, this means interpreting a carriage return as end-of-line, echoing a line feed, and returning \n. For **getc**, this means simply ignoring carriage returns, because they appear as a pair with newlines in disk files.

FILES

stdio.h, stdio.c, fileio.c

SEE ALSO

fopen(subr), putc(subr)

LIMITATIONS

On input to **getchar**, when output is re-directed to a file, tabs may not be expanded properly. This is due to an obscure CP/M bug, where it loses track of the column position unless a line-feed is output to the terminal via "console output," while getchar must use "direct console I/O" to bypass output redirection.

Carriage returns without following line-feeds in disk files should probably not be ignored by **getc**, so as to allow over-printing.

NAME

fopen — Buffered File Opening and Closing

SYNOPSIS

```
FILE *filep;
 ...
filep = fopen(filename, mode);
 ...
fclose(filep);
```

DESCRIPTION

The **fopen** subroutine attempts to open/create the file with the given filename, for the given mode (either "r", "w", "a", "rb", "wb", or "ab"). If successful, it returns a pointer to a dynamically allocated (see malloc(subr)) **FILE** buffer structure. If not, it returns the defined value **NULL**.

The filename must be a normal null-terminated C string, using CP/M filename format of **x:name.ext**. The drive letter may be omitted if the current working directory is desired.

The **mode** for the fopen is also specified by a string. The first letter of the string specifies read-only, write-only, or append-only. The second letter, if **b**, specifies a binary file, so embedded control/z (text end-of-file) characters are ignored, and no other translations are performed.

Once opened, the returned **FILE** pointer can be passed to getc (if open for "r" or "rb") or putc/fprintf (if open for "w", "wb", "a", or "ab").

When processing is complete, the FILE pointer should be closed, with fclose, thereby writing to disk any partial buffer-full, and releasing storage reserved for the internal FILE buffer structure.

When opening for write-only, the file is created. It must not already exist. When opening for append-only, the file may exist, but it will be created if necessary.

FILES

stdio.h, fileio.c, fopen.c

SEE ALSO

getc(subr), malloc(subr), putc(subr)

LIMITATIONS

It might be convenient to provide a mode which would allow overwriting an existing file.

NAME

main — Hidden Main Routine; Exit Routine

SYNOPSIS

```
c_main();

main(argc, argv)
char **argv;

exit(exit_status);
```

DESCRIPTION

The routines **c_main** and **exit** are provided in the file **cmain.c**. **c_main** is where execution should actually begin for all C programs. It builds an argument list following the UNIX conventions, and then calls the user-provided **main** routine with an argument count (**argc**) and a pointer to an array of string pointers (**argv**).

The arguments are constructed from the information in the CP/M "default buffer area" (0x80 hex), after expansion of filename wildcards (? and *). The zeroth argument (**argv[0]**) per UNIX convention represents the command name itself, but will always be - because CP/M does not record the command name in the buffer area.

The argument count (**argc**) includes one for the command name, and hence **argv[argc-1]** is actually the last valid argument. **Argv[argc]** is always a null pointer.

Besides expanding filename wildcards (see wildcard(info)), **c_main** also removes quotes (both ' and ") and backslashes (\), except within parentheses. Text within parentheses is passed as is, including the parentheses themselves, as a single argument. If some arguments contain wildcards, and no matches are found for any of them, **c_main** prints 'No match' and aborts. Wildcard characters within quotes are not expanded.

The **exit** routine should be called at completion of the program, with a status value. **Exit** records this value for use by the C/NIX shell (see csh(cmnd)) by calling BDOS function 108 (ignored by normal CP/M), and then jumps to zero for a warm start. If the exit status value is negative, the shell will interpret this as an error exit, and abort any current command file or sequence.

FILES

cmain.c

SEE ALSO

csh(cmnd), wildcard(info)

LIMITATIONS

When wildcarded filenames are expanded, the resulting arguments should probably be sorted alphabetically, instead of in directory order as they are now.

NAME

malloc — Dynamic Memory Allocation and Release

SYNOPSIS

```
struct my_rec *ptr;
  ...
ptr = malloc(sizeof(struct my_rec));
  ...
free(ptr);

mp = compress();
```

DESCRIPTION

These routines provide for dynamic memory allocation and release for arbitrary sized C structures or arrays. The **malloc** routine takes a size in bytes, and returns a pointer to the start address of an allocated area. When done with the area, the space can be released with **free**.

If **malloc** cannot allocate sufficient space without running into the C stack, it will print an error message and return a NULL pointer.

The **compress** routine attempts to compress the "heap" from which **malloc** allocates areas, and then returns a pointer to a structure describing the "heap," allowing it to be saved en masse (see **malloc.h**).

NOTES

Returning an area to **free** which was not allocated with **malloc** is disastrous.

The area allocated is NOT initialized to zeros.

Malloc uses a word in front of each area to indicate its length, as well as whether it is free or in use. These allow malloc to iterate through all areas, and locate a large-enough contiguous free chunk. It is designed to be particularly efficient if memory is used approximately in a last-allocated, first-freed order.

NAME

putc — Standard Buffered Character Output

SYNOPSIS

```
FILE *filep;
  ...
putc(c, filep);
  ...
fflush(filep);

putchar(c);
```

DESCRIPTION

The subroutines **putc** and **putchar** provide standardized means for character output, either buffered to a disk file, or to the console output.

The **FILE** pointer passed to **putc** must have been the result of an **fopen**, or be one of the standard file pointers **stdout** or **stderr**.

Putc to a disk **FILE** pointer adds a character to an internal buffer, and then actually writes it to the disk when the buffer is full. The **fflush** subroutine may be used to flush partial buffer-fulls (on CP/M or C/NIX, it can only flush in units of the basic 128-byte sector). An **fclose** must be performed when done, so that the final partial buffer-full is written, terminated by the CP/M text end-of-file indicator (control/Z) (see fopen(subr)).

Putc(c, stdout) is equivalent to **putchar(c)** (see below). **Putc(c, stderr)** puts the character out using the "direct console I/O" BDOS call, thereby bypassing normal C/NIX output re-direction (but see >& in ioredir(info)).

The putchar subroutine puts out the character using the "console output" BDOS call. If output has been re-directed, the character will actually go to a file.

For all of the "put" routines, if the character is the C newline ('\n'), then a carriage return ('\r') is put out first (except if the file was opened in "binary" mode — see fopen(subr)).

FILES

stdio.h, stdio.c, fileio.c

SEE ALSO

fopen(subr), ioredir(info)

NAME

strutils — Standard String Utilities

SYNOPSIS

```
if (strany(c, str)) ...

if (strcmp(str1, str2) > 0) ...

cp = strcpy(to, from);

if (streql(str1, str2)) ...

l = strlen(len);

mvbytes(from, to, num);
```

DESCRIPTION

These routines provide standard means for manipulating null-terminated strings in C. All but **mvbytes** are based on the UNIX equivalents.

Strany returns non-zero if the given character appears anywhere within the given string.

Strcmp returns greater than zero, equal to zero, or less than zero according to whether **str1** is later, the same, or earlier lexicographically than **str2**. Lexicographic order means that if one is a prefix of the other, then it is considered earlier (or less). When the strings are alphabetic only, the order is simply conventional alphabetical order.

Strcpy copies characters, up to and including the null terminator, from its second argument **from** to a buffer area pointed to by its first argument **to**. The buffer must be long enough to hold the copied string. Strcpy returns a pointer to the null at the end of the copy, allowing a cascading series of **strcpys** to do concatenation, as follows:

```
/* Concatenate 'first' and 'second' into 'buf' */
strcpy(strcpy(buf, first), second);
```

Streql returns non-zero if **str1** and **str2** are identical strings.

Strlen returns the length of **str**, not counting the null-terminator.

Mvbytes copies **num** bytes from an area **from** to an area **to**. **Mvbytes** does NOT stop at null-terminators.

NOTES

The **strcpy** routine familiar from UNIX returns a pointer to the BEGINNING of the copy, not the end as it does here.

The order of parameters for **strcpy** and **mvbytes** are the opposite of one another.

FILES

strutils.c

. notation 18
.. notation 18
... notation 38
/ switches 21
[notation 38
| notation 38

aborting command file 33
ambiguous file name 24
arguments, limits on 33
autostart command 8

backup, problems with 23
bad directory message 21
batch files 30
bdos subroutine 48, 49
begin, how to 4
bottom-up 47
bye command 9, 25, 35

C character output routines 54
C character read routines 50
C file routines 51
C main program 52
C memory allocation routines 53
c source files 8
C string functions 55
C system call 49
C/NIX, exiting from 35
C/NIX, first use of 8
C/NIX, installing 7, 8
C/NIX, starting 6
cat command 25, 26
cd(cmnd) ..... see chdir(cmnd)
change working directory 27
changing defaults 16
character output routines 54
character read routines 50
chdir command 25, 27
chmod command 25, 28
cnixinit.sub file 8, 46
cnixutil.pre file 9
cold boot 8
command file, aborting 33
command files 12, 30
command files, echo in 32
command format 30
command line, limits on 33
command processor 30
command search path 12, 30
command: echo 32
command: cat 26
command: cd ..... see chdir(cmnd)
command: dir 40
command: era 12, 45
command: exit 9
command: grep 7
command: help 7, 9, 12
command: man 7, 38
command: pwd 10, 27
command: ren 44
command: rm 12
command: rmdir 42
command: set 46
command: type ..... see cat(cmnd)
commands 25
commands over subdirectories 47
commands, grouping 30
comments to author 4
compress subroutine 48, 53
concatenating files 26
conditionals, how to do 32
confirm required, changing 16
console paging 46
copying a file 29
copying between directories 29
copying to existing file 29
cp command 25, 29
csh command 25, 30
current directory 18
current working directory 27
c_main subroutine 52

defaults, changing 16
delete a file 45
description 38
DESPOOL 23
dir command 25, 40
directories, copying between 29
directories, hierarchical 10
directories, implementation of ... 42
directory listing 40
directory sizes, display 40
directory structure 18
directory tree, walking 47
directory, changing 27
directory, current 18
directory, making 42
directory, printing current 27
directory, top level 10, 18

Index

diskettes, restrictions on changing 9
display directory sizes 40
display file sizes 40
display working directory 27
double spaced output 15
drives for temp files, changing .. 16
dynamic memory allocation 53

echo command 25, 32, 34
echo in command files 32
era command 12, 25, 45
erase a file 45
escape characters 21
examples 38
executable flag 40
exit command 9, 25, 35
exit status 33, 52
exit subroutine 52
exiting C/NIX 35

faster execution 43
fclose subroutine 48, 51
features 5
features, introduction to 10
fflush subroutine 48, 54
file flags 28
file modes 40
file names, display 40
file routines 51
file search 28
file sizes, display 40
file, copying 29
file, copying to existing 29
file, erasing 45
file, invisible 28
file, renaming 44
filenames 18, 24
files, batch 30
files, command 30
files, concatenating 26
files, missing 22
files, searching 36
files, showing 26
files, submit 30
filters 20
finding a file 28
first use 8
flag, read-only 28
flag, SYS 28
flag, writeable 28
flags, file 28
fopen 51
fopen subroutine 48, 51
format of commands 30
free subroutine 48, 53

getchar subroutine 48, 50
getting started 3
grep command 7, 25, 36
grouping commands 30

help command 7, 9, 12, 25, 38
help drive, changing 16
hierarchical directories 10, 18
how to begin 4

I/O redirection 11, 13, 30
I/O redirection, problems with ... 22
input redirection 13
installing C/NIX 7, 8
introduction 5
introduction to features 10
invisible file 28
invoking the shell..... 30
ioredir 13

leaf 47
leaving C/NIX 35
leaving shell 35
limitations 38
limits on command line 33
list file names 40
list file sizes 40
ls command 25, 40

main program 52
main subroutine 48
making directory 42
malloc subroutine 48, 53
man command 7, 25, 38
manual pages 38
manual pages, notation for 38
manual, how to use 3
manual, organization of 3
manual, starting with 4
memory allocation routines 53
memory requirements 5
Microsoft BASIC 23
missing files 22

mkdir command 25, 42
mkrel command 25, 43
modes, file 40
move a file 44
mv command 25, 44
mvbytes subroutine 48, 55

names of files, display 40
notation for manual pages 38
notes 38

organization of manual 3
output redirection 13

page relocatable programs 43
paging 46
paging output 9, 12
paging, changing 16
patching defaults 16
pathname 10, 18
pathnames, searching all 47
pattern matching 36
pipes 20, 30
pipes, problems with 22
power off 9
pre files 43
print working directory 27
problems 4
putc subroutine 48, 54
putchar subroutine 48, 54
pwd command 10, 25, 27

quoting 21

read only flag 28, 40
redirection, I/O 11, 13, 30
redirection, problems with 22
regular expression 36
relocatable programs 43
removing directory 42
ren command 25, 44
rename a file 44
requirements, disk 5
requirements, memory 5
restrictions on changing diskettes 9
rm command 12, 25, 45
rmdir command 25, 42

SAVE (unsupported) 23
saving program output 13
screen paging of output 9, 12
screen paging, changing 16
search for files 28
search path 12, 30
searching all directories 47
searching files 36
see also 38
set command 46
shell, exiting from 35
shell, invoking 30
size of files, display 40
slow prompts 22
source code 8
special characters 21
spoolers 23
starting C/NIX 6
starting with manual 4
startup command file 8, 46
status at exit 33
strany subroutine 48, 55
strcmp subroutine 48, 55
strcpy subroutine 48, 55
streql subroutine 48, 55
string functions 48, 55
strlen subroutine 48, 55
strutils 55
subdirectories 10, 18
subdirectory sizes, display 40
submit command 25
submit files 12, 30
subrs 48
switch / 21
switches 38
synopsis 38
SYS flag 28
sys flag required, changing 16
system call 49

tabs, problems with 23
top level directory 10, 18
top-down 47
trouble 22
type command ..... see cat(cmnd)

UNIX-like features 10
user number, changing maximum 16
user numbers 42
using manual 3

verbose flags, changing 16
verbose mode, setting 46

walk command 11, 18, 25, 47
walking directory tree 47
wildcard 24
working directory 18, 27
write protection 28
writeable flag 28

XSUB (not supported) 14, 23