

GNO Shell User's Manual

By Tim Meekins, Albert Chin, and Jawaid Bazyar
Edited by Andrew Roughan

Table of Contents

Chapter 1	Getting started with the GNO Shell	3
	Introduction.....	3
	Customizing the Shell Environment.....	3
	Invoking gsh.....	5
Chapter 2	Interacting with the GNO Shell	7
	Executing Commands.....	7
	Command-line Editor.....	7
	Command Input.....	8
	Command Editing.....	8
	History Editing Keys.....	9
	Command, Filename and Variable completion.....	10
Chapter 3	Using the GNO Shell more productively	13
	What does this command do?.....	13
	Option Arguments.....	13
	Entering Multiple commands.....	15
	Using Aliases as Shorthand.....	15
	Redirecting Input and Output.....	16
	Pipelines.....	16
	Background Execution of Commands.....	17
	Job Control.....	19
	Working with Pathnames.....	21
	Pathname Expansion.....	22
	Quoting Special Characters.....	23
	How gsh Finds a Command.....	24
Chapter 4	Built-in Commands	25
	Built-ins vs EXE Commands.....	25
	Shell Commands.....	25
	Kernel Commands.....	29
	Environment Commands.....	30
Chapter 5	Shell Variables	33
	Using shell variables.....	33
	Scope.....	33
	Description of Pre-defined Shell Variables.....	33
	Accessing shell variables.....	35
Appendix A	Sample gsh session	39
Appendix B	Prefix Conventions	45
Appendix C	Errors	46
Appendix D	Signals	49
Appendix E	Non-Compliant Applications	50
Appendix F	Termcaps	51
Glossary	53
Index	55

Chapter 1

Getting started with the GNO Shell

"Computer operating systems are among the most complex objects created by mankind..."
Douglas Comer, Operating System Design, The XINU Approach

Introduction

The GNO shell is an integral part of the GNO Multitasking Environment (GNO/ME). The GNO shell provides the interface between the user and the GNO Kernel. While both work together, the jobs they perform are quite different. This manual documents the functions of the shell.

The user interacts with the shell through a command-line interface. Command-line interfaces provide a unique way of interacting with the operating system. Unlike GUIs (Graphical User Interfaces), with which you are already familiar with by using programs such as the Finder and ShrinkIt! GS, all commands are typically entered using the keyboard. The shell interprets commands and passes them to the kernel for control and execution.

The command-line interface will be unfamiliar to some people. However, once the command-line interface has been mastered, the user should have no difficulty using any current or future GNO applications. Those of you already familiar with Unix interfaces, such as the C shell, Bourne shell, and Korn shell, or the ORCA shell on the Apple IIGS, will begin to realize the advantages which GNO/ME is able to provide.

The way this manual is presented allows the complete beginner to simply work through the chapters in a chronological order. Chapter 2 familiarises the user with entering basic commands whereas the more powerful GNO/ME features are introduced in Chapter 3. Chapter 4 documents the commands which are built into the GNO Shell and Chapter 5 explains shell variables which give the user control over how their installation functions.

Customizing the Shell Environment

When **gsh**, the implementation of the GNO Shell, is executed, it reads in and processes the *gshrc* file. This file contains start-up instructions for the shell, which can be used to customize the operation of the shell and other aspects of the system. It is created by the GNO Installer during the installation process.

The following is a sample *gshrc* file (line numbers have been added for convenience):

###	line #1
#	line #2
# GNO 2.0 gshrc file	line #3
#	line #4
###	line #5
#	line #6
# Initialize our environment	line #7
#	line #8
set path=":hard:gno:bin :hard:gno:usr:bin"	line #9
set prompt="[%h] %S%t%s %C> "	line #10
set home=":hard:gno:user:root"	line #11
set term=gnocon	line #12
export path prompt home term	line #13
setenv history=100 savehist=25	line #14

```

### line #15
# line #16
#Set up standard prefixes for utilities. line #17
# line #18
### line #19
prefix 2 :software:orca:libraries line #20
prefix 3 :software:orca line #21
prefix 4 :software:orca:shell line #22
prefix 5 :software:orca:languages line #23
prefix 6 :software:orca:utilities line #24
prefix 7 :tmp line #25
### line #26
# line #27
# Set up prefixes for Orca2.0(tm)'s benefit line #28
# line #29
### line #30
prefix 13 :software:orca:libraries line #31
prefix 14 :software:orca line #32
prefix 15 :software:orca:shell line #33
prefix 16 :software:orca:languages line #34
prefix 17 :software:orca:utilities line #35
alias ls 'ls -CF' line #36
alias dir 'ls -al' line #37
alias cp 'cp -i' line #38
alias rm 'cp -p rm' line #39
alias mv 'cp -p mv' line #40
setenv usrman='/usr/man' line #41
set fignore='.a .root .sym' line #42
alias zcat 'compress -cd' line #43
setenv pager=less line #44
setenv less=-e line #45
set nonewline=1 line #46
# line #47
# Move to home directory line #48
# line #49
cd line #50

```

When you install GNO/ME, the GNO installer knows where to find the GNO utilities and any ORCA utilities you may have. Unfortunately it does not know where all the other utilities and applications that you may wish to use are located. It is therefore necessary to edit the setup file in order to tell the GNO shell where these programs are on your hard disk.

The setup file, *gshrc* is located in the /usr directory of the path where you installed GNO/ME. You can use any text editor from the desktop to edit the *gshrc* file, or if you are already familiar with the editor *vi* you can use this utility after launching the GNO kernel.

Line 9 is the statement that we are concerned with. **Hard** represents the name of your particular hard drive volume where you have installed GNO/ME.

```
set path=":hard:gno:bin :hard:gno:usr:bin" line #9
```

You will see that spaces have been inserted between pathnames. The space is the pathname separator and the colon has been used as the path delimiter for this specific variable, *PATH*. As an exercise, add your system directory to this statement. Line 9 should now look like this:

```
set path=":hard:gno:bin :hard:gno:usr:bin :hard:system"
```

What you have just done allows the GNO shell to find the **Finder** application. Now go ahead and add any pathnames that hold utilities or applications that you will use frequently from GNO/ME. It should also be noted that it is possible to have more than one pathname containing EXE, SYS16, or EXEC files; this is impossible under ORCA. The PATH variable is discussed thoroughly in Chapter 5.

For now, the remaining lines of the *gshrc* file do not need editing. As you gain an understanding of the system you may wish to make further changes to the *gshrc* file. Make sure you save the file before you exit the editor.

It is possible to modify these instructions while the GNO shell is active, but any changes will be lost upon exiting **gsh**. If you wish the changes to remain effective for the next session you must add them to the *gshrc* file.

By customizing the *gshrc* file it is possible to make the GNO environment more like UNIX, the ORCA environment, or something completely different. Customization of the GNO environment leads to greater user productivity.

Invoking gsh

GNO/ME can be launched from a program launcher, such as the System 6.0 Finder. Launch the GNO Kernel program, **kern** by double clicking on it. The GNO kernel automatically executes the supplied GNO shell, **gsh**.

The prompt, "[1] 2:00am root %", indicates that **gsh** is ready to receive input from the keyboard.

To start a new **gsh** from the command-line simply type **gsh**. If multiple copies of the **gsh** process are undesirable, use the command **source gsh** instead. This is useful for testing changes made to the *gshrc* file. **Source** is a built-in command which is discussed in Chapter 4 **Shell commands**.

Chapter 2

Interacting with the GNO Shell

Executing Commands

A command consists of two parts: a name and its arguments. The command name is the name used to start the command. The name is usually the name of a file which can be executed. The only exceptions are commands which are built-in to the shell. These commands are documented in Chapter 4 **Built-In Commands**. Any shell utility command with a filetype of EXE, SYS16, or EXEC can be executed in this fashion. The command name must be separated from the command arguments with a space.

The command arguments are parameters that the command takes as data to work with (In Applesoft BASIC, "HELLO WORLD" would be an argument for the PRINT command). Command arguments are separated from each other with a space. Note that although arguments extend the usefulness of a command, not all commands have arguments. Any arguments entered after the command will be passed by the shell to the program when it starts executing.

The examples below use the following commands:

qtime	displays time in English text.
echo	prints arguments to the screen.

Examples:

```
% qtime
It's almost five.
% echo II Infinitum
II Infinitum
```

At the simplest level the user enters commands to the shell by typing them on the keyboard. **Gsh** includes a command-line editor to help the user enter and edit commands. The editor also provides a way to modify and execute previous commands. Additionally the editor can help complete the names of commands, filenames and variables.

Command-line Editor

Below is a complete description of the functions of the command-line editor with short examples depicting how each editing key works.

Throughout the examples the underline character, "_", will be used to represent the current cursor position. In addition, "OA" is used to represent the Open Apple key and the term *word* is used to indicate a string of characters consisting of only letters, digits, and underscores. To the right of a editing key entry is the `bindkey` function name which is used to remap editing functions to new keys. This information is included for reference purposes only. See Chapter 4 **Shell Commands** for more information on the `bindkey` command.

It should be pointed out that at this stage that the user should not be concerned with what the actual commands used in the examples do, rather the user should concentrate on how the command-line editor functions work.

Command Input

These command-line editor keys deal with entering text directly on the command-line.

RETURN	<p>newline</p> <p>The return key is used to terminate line input. gsh then interprets the command on the line and acts accordingly. The position of the cursor on the command-line does not matter.</p> <p>Before</p> <pre>% echo foo bar (RETURN)</pre> <p>After</p> <pre>foo bar % _</pre>
CTRL-D	<p><no bindkey name></p> <p>Exits gsh if it was the first character typed on the command-line. If there are still jobs running in the background or stopped, gsh will display "There are stopped jobs." If you press CTRL-D a second time without an intervening command, gsh will terminate all the jobs in the job list and exit.</p>
CTRL-R	<p>redraw</p> <p>Moves to the next line and re-displays the current command-line. Use this to redraw the current line if the screen becomes garbled.</p>
CTRL-L	<p>clear-screen</p> <p>Clears the screen, moves the cursor to the top line, and redraws the prompt and any command-line that was in the process of being edited.</p>

Command Editing

These command-line editor keys allow editing of the command-line text.

CTRL-B LEFT-ARROW	<p>backward-char</p> <p>Moves the cursor one character to the left. You cannot move past the first character on the line. If so, gsh will output an error beep.</p> <table> <tr> <td>Before</td><td>After</td></tr> <tr> <td>% print foo bar (LEFT-ARROW)</td><td>% print foo bar</td></tr> </table>	Before	After	% print foo bar (LEFT-ARROW)	% print foo bar
Before	After				
% print foo bar (LEFT-ARROW)	% print foo bar				
CTRL-F RIGHT-ARROW	<p>forward-char</p> <p>Moves the cursor one character to the right. You cannot move past the last character on the line. If so, gsh will output an error beep.</p> <table> <tr> <td>Before</td><td>After</td></tr> <tr> <td>% print foo bar (RIGHT-ARROW)</td><td>% print foo bar</td></tr> </table>	Before	After	% print foo bar (RIGHT-ARROW)	% print foo bar
Before	After				
% print foo bar (RIGHT-ARROW)	% print foo bar				
DELETE	<p>backward-delete-char</p> <p>Deletes the character to the left of the cursor. You can delete up to the first character on the command-line.</p> <table> <tr> <td>Before</td><td>After</td></tr> <tr> <td>% print foo bar (DELETE)</td><td>% print o bar</td></tr> </table>	Before	After	% print foo bar (DELETE)	% print o bar
Before	After				
% print foo bar (DELETE)	% print o bar				
CLEAR CTRL-X	<p>kill-whole-line</p> <p>Deletes all characters on the command line and positions the cursor after the prompt.</p> <table> <tr> <td>Before</td><td>After</td></tr> <tr> <td>% print foo bar (CTRL-X)</td><td>% _</td></tr> </table>	Before	After	% print foo bar (CTRL-X)	% _
Before	After				
% print foo bar (CTRL-X)	% _				

CTRL-Y	kill-end-of-line Deletes all characters from the cursor to the end of the command-line.
	Before % print <u>f</u> oo bar (CTRL-Y)
	After % print f_
CTRL-D OA-D	delete-char Deletes the character under the cursor.
	Before % print fo <u>o</u> bar (OA-D)
	After % print fo_bar
CTRL-A OA-<	beginning-of-line Moves the cursor to the beginning of the line.
	Before % print <u>f</u> oo bar (OA-<)
	After % print foo bar
CTRL-E OA->	end-of-line Moves the cursor to the first position past the last character on the line.
	Before % print foo ba <u>r</u> (OA->)
	After % print foo bar_
OA-RIGHT-ARROW	forward-word Moves the cursor right to the last character of the current word.
	Before % print foo ba <u>r</u> (OA-RIGHT-ARROW)
	After % print fooo bar
OA-LEFT ARROW	backward-word Moves the cursor left to the beginning of the current word.
	Before % print <u>f</u> oo bar (OA-LEFT ARROW)
	After % print <u>f</u> oo bar
	Before % print foo <u>b</u> ar (OA-LEFT ARROW)
	After % print fooo bar
OA-E	toggle-cursor Toggles input mode between insert and overstrike. Overstrike mode is distinguished by a solid inverse cursor and insert mode by a blinking '_' cursor. In overstrike mode, any characters that are typed directly over-write those characters below the cursor. In insert mode, the characters typed are inserted before the character below the cursor.

History Editing Keys

These command-line editor keys allow access to previously entered commands. The GNO shell automatically keeps track of previous commands in what is called a history buffer.

The maximum number of command-lines saved in the history buffer is determined by the shell variable `HISTORY`. A default value for this variable is set in the `gshrc` file that the GNO Installer creates. The lines saved to the history buffer are kept between sessions. That is, when you exit **gsh**, `$SAVEHIST` command-lines are saved to the `history` file in your `$HOME` directory. When **gsh** is invoked again, all command-lines saved in the history buffer will be available using history editing keys. See Chapter 5 **Description of Predefined Shell Variables** for more information on the `HISTORY` and `SAVEHIST` variables.

CTRL-P **up-history**

UP-ARROW Fetches the previous command-line. If the current command-line is the first line in the history buffer, the next line fetched will be an empty command-line. If invoked again, the last line in the history buffer will be displayed.

Before	After
% print foo bar	
% echo Apple II	
% echo GNO/ME	
% _ (UP-ARROW)	% echo GNO/ME_
% echo GNO/ME_ (UP-ARROW)	% echo Apple II_

CTRL-N **down-history**

DOWN-ARROW Fetches the next command-line. If the current command-line is the last command line in the history buffer, the next line fetched will be the first command-line in the history buffer.

Before	After
% print foo bar	
% echo Apple II	
% echo GNO/ME	
% _ (DOWN-ARROW)	% print foo bar_
% print foo bar_ (DOWN-ARROW)	% echo Apple II_

Command, Filename and Variable completion

These command-line editor keys can be used to complete filenames, commands and variables.

CTRL-D **list-choices**

Lists commands and pathnames that match the current word.

Before	(CTRL-D)
% print foo_ bar	
After	
foo.c fool foo.m	
% print foo_ bar	

TAB **complete-word**

Command, pathname and variable completion. If the cursor is positioned on the first word of the command-line, command pathname is performed, else pathname or variable completion is performed. The word is expanded to the closest matching command, pathname or variable. Characters are appended up to the point that they would cause more than one. If a complete pathname results for pathname completion, **gsh** appends a "/" if the pathname is a directory; otherwise, it appends a space.

Before	After
% ca _ (TAB)	% cat _
% mo _ foo.c (TAB)	% more _ foo.c
% more fo_ (TAB)	% more foo_
% echo \$TERMC_ (TAB)	% echo \$TERMCAP_

Note that if there is more than one match for the partial command, **gsh** will sound a beep on the speaker. You can use the CTRL-D (**list-choices**) command to see the list of possible matches, and should either finish entering the command manually or type enough additional characters to guarantee a unique match.

If the \$FIGIGNORE variable is set, **gsh** ignores filenames (when doing completion) that end with any of the suffixes in \$FIGIGNORE. See Chapter 5 **Description of Pre-defined Shell Variables** for more information regarding the \$FIGIGNORE variable.

Other ways of entering commands

Terminal Input

An example involving the connection of a terminal; will be dealt with in Chapter 3 **Redirecting Input and Output** but it is necessary to mention here that when using **gsh** over a terminal, some keystrokes must be slightly modified. This is because there are no terminals that can transmit the OA key. Instead, a two-key sequence must be used which replaces OA with ESC. For example, instead of pressing OA-E to toggle insert mode, you can type ESC-E over a terminal to do the same thing.

If you will be using terminals seriously then you should install the Remote Access package.

Script File

While you would normally type commands on the command-line, you can also store a series of often used commands in a file. A file containing such a series of commands is called a script. A script is normally created by using a text editor.

By typing the name of the script file, the shell will execute it, line by line, as if you had typed each command separately. The *gshrc* file presented in Chapter 1 **Customizing the Shell Environment** is an example of a script file.

Chapter 3

Using the GNO Shell more productively

"And then one day, hooray! Another way for gnomes to say hooray!"
Syd Barret, The Gnome

What does this command do?

If you are unfamiliar with what a particular command actually does or what arguments it accepts, you can check quickly by using the electronic manual. GNO/ME includes a utility called **man** which displays the manual pages for a command whose name you supply as an argument. The **man** utility uses another utility called **more** to actually display the pages nicely on the screen. While both of these utilities have electronic manual entries, we have included the printed manual pages in the GNO/ME package to get you started.

Option Arguments

As mentioned in Chapter 2 **Executing Commands**, arguments are passed to a command to extend its usefulness. The arguments presented in the last chapter were words, such as foo, bar and foo.c. Standards exist under unix for programs to accept command-line option arguments. Option arguments (as the name suggests) are optional. There are two standards, short options and long options. Short options are characters that represent commands, whereas long options contain the entire option name.

Consider the following output of the CATALOG command from ProDOS:

/DEV NAME	TYPE	BLOCKS	MODIFIED	CREATED	ENDFILE	SUBTYPE
FINDER.DATA	\$C9	1	21-OCT-91 22:38	14-APR-90 18:24	260	
FINDER.ROOT	\$C9	1	22-OCT-91 17:12	6-OCT-91 15:40	82	
GENESYS	DIR	1	21-OCT-91 23:37	25-APR-91 15:46	512	
GSBUG	DIR	1	21-OCT-91 23:38	19-JUL-90 16:48	512	
MERLIN	DIR	2	22-OCT-91 2:50	30-APR-91 20:21	1024	
LIFEGUARD	\$B3	73	4-SEP-87 4:51	25-DEC-89 20:22	36608	
ORCA	DIR	2	22-OCT-91 17:12	14-SEP-89 18:27	1024	
GNO	DIR	2	22-OCT-91 17:12	13-AUG-91 16:36	1024	
FAST.ANIM	DIR	2	21-OCT-91 23:44	11-MAY-91 10:50	1024	
MICOL	DIR	2	22-OCT-91 3:10	14-JAN-90 2:46	1024	
SRC	DIR	1	21-OCT-91 23:44	7-AUG-91 20:30	512	
NIFTYLIST.V3.3	DIR	2	21-OCT-91 23:44	29-JUL-91 4:04	1024	
MCSRC	DIR	1	21-OCT-91 23:45	7-AUG-91 20:34	512	

BLOCKS FREE:43923 BLOCKS USED:21185 TOTAL BLOCKS:65108

It is impossible to get any variation in the format of this output. While the GNO/ME utility **ls** serves the same purpose as the command CATALOG from Applesoft BASIC, it has a wide number of options which can tailor the output to specific needs. Here is how **ls** can be used to give similar output to the CATALOG command:

```
[1] 2:13am root % ls -l
:dev
total 45k
drw--rd 0000 dir          512 Oct 21 23:45 1991 MCSrc
drw--rd 0000 dir        1024 Oct 21 23:44 1991 NiftyList.v3.3
```

```

drw--rd 0000 dir      1024 Oct 21 23:44 1991 fast.anim
drw--rd 0000 dir      512 Oct 21 23:37 1991 genesys
drw--rd 0000 dir      1024 Oct 22 17:29 1991 gno
drw--rd 0000 dir      512 Oct 21 23:38 1991 gsbug
drw--rd 0000 dir      1024 Oct 22 02:50 1991 merlin
drw--rd 0000 dir      1024 Oct 22 03:10 1991 micol
drw--rd 0100 dir      1024 Oct 22 17:28 1991 orca
drw--rd 0000 dir      512 Oct 21 23:44 1991 src

```

The **-l** short option argument tells **ls** to format the output in long format. **ls** supports only short options. If **ls** did support long options, the above command could be changed to **ls +format=long**. This is clearly more descriptive of what function **ls** will perform. For users new to the UNIX environment, long format options are more user-friendly. However, advanced UNIX users prefer short options because of their brevity.

As indicated above, **ls** has a wide number of options available to format the output. Use the command "**ls -?**" to get a short list of these options. It is left as an exercise for the user to discover how these options affect the output of **ls**. For a complete description of the **ls** command and its options use the command **man ls**.

As an example of the usage and importance of long options, the following is the result of the **+help** option given to the **coff** utility. Note the use of both short and long options:

```
coff [-OPTIONS] filename [segment..] [loadsegment..]
```

OPTIONS	DESCRIPTION
-v [+version]	display coff's version number
-D [+default]	disable default options
-d [+asm]	dump segment body in 65816-format disassembly
-T [+tool]	interpret Toolbox, GS/OS, ProDOS, ROM calls
-x [+hex]	dump segment body in hex (can be used with '+asm')
-l [+label]	print expressions using labels (default is offsets)
-t [+infix]	display expressions in infix form
-p [+postfix]	display expressions in postfix form (default)
-m [+merlin]	format of '+asm' to use merlin opcodes (default)
-o [+orca]	format of '+asm' to use orca/m opcodes
-a [+shorta]	assume 8-bit accumulator for disassembly
-i [+shorti]	assume 8-bit index registers for disassembly
-s [+header]	dump segment headers only
-n [+noheader]	do not print segment headers
-f [+nooffset]	do not print offset into file
-h [+help]	print this information, then quit
filename	name of file to dump
[segment]	names of segments in file to dump
[loadsegment]	names of load segments in file to dump

The long options are much more descriptive, and provide a very easy way to remember options of programs. If an option passed to a shell utility program is not understood by that program, you will generally receive an error message stating that the option is not understood. If the program is user-friendly, a brief list of supported options will also be displayed.

Entering Multiple commands

It is possible to give multiple commands to the GNO shell for processing. To execute multiple commands, place a semi-colon, ";", between them. The commands will be executed sequentially in the order they are entered on the command-line. Take care not to exceed the 1024 character command-line buffer. It is possible to execute multiple commands at the same time, this feature is discussed in Chapter 3 **Background Execution of Commands**.

To run the **echo** command and the **ls** command in succession, enter the following on the command line:

```
% echo Running ls ; ls -l
```

The output of the preceding command will display the string "Running ls" followed by the output of the "ls -l" command. This method of running several commands in succession is used often to save typing.

Using Aliases as Shorthand

gsh provides a built-in command, **alias**, which allows any command you would type on the command-line to be renamed. You are not limited to renaming a single command name. Rather, you could rename an entire command-line, which could allow you to use the name "backup" to execute the command "backup +source /system +destination /tape.drive". The **alias** command is also a very powerful means of customizing your GNO environment to emulate other computing environments.

To emulate the ORCA environment, the following aliases could be entered into your *gshrc* file, or a script called *orca.alias* that *gshrc* would run:

```
alias copy cp
alias cat "ls -l"
alias catalog "ls -l"
alias move mv
alias rename mv
alias delete rm
alias type cat
alias prefix cd
alias create mkdir
```

If you alias a string containing multiple words, you must enclose the string in quotes, as done for the catalog alias. **gsh** interprets the string as one value. If you do not include both the opening and closing quotes, the alias command will notify you of your error.

You can view any alias' that are set by entering the **alias** command without any arguments. The setting of a particular alias can be viewed by entering one argument consisting of the name of the alias to view.

If you wish to remove an alias, use the command **unalias** with the aliased name as the argument. To remove the aliases from the *orca.alias* file given above, you could do the following:

```
% unalias copy cat catalog move rename delete type prefix create
```

Unlike the **alias** command, the **unalias** command can take multiple arguments. See Chapter 5 **Built-in Commands** for further discussion of the **alias** and **unalias** commands.

Redirecting Input and Output

Most shell utilities write their output to the screen. However, under GNO/ME, like ORCA, it is possible to redirect that output to a file or a GS/OS device. The output of the `ls` command above was imported into this manual by redirecting it to a file. In addition to redirecting the output of a shell utility, it is also possible to redirect the input of that utility. Consider the following `gsh` session:

```
[1]% echo this is a test
this is a test
[2]% echo this is a test > file1
[3]% cat file1
this is a test
[4]% cat < file1
this is a test
```

In the example above, `cat` takes input from *standard input*. In command 3 above, `cat` takes as an argument the filename `file1` and writes the contents of that file to *standard output*. Where no filename argument is given, `cat` reads input from standard input and writes the output to standard output.

In the case of command 4 above, `cat` contains no arguments and therefore reads from standard input. However, `gsh` interprets the "<" redirection operator and opens the file `file1` for use as standard input. Therefore, `cat` will take its input from `file1`, even though it thinks it is reading input from standard input. This input redirection is transparent to the utility, making it work with most shell utilities.

Command 2 above created a new file called `file1`. If this file had existed prior to the command then it would have been erased. It is possible to append output to the end of the file by using the ">>" redirection operator. Consider the following `gsh` session:

```
[5]% echo second line >> file1
[6]% cat file1
this is a test
second line
```

Output that is sent to *standard error*, can also be redirected. The ">&" operator redirects standard error to a file and ">>&" appends standard error to the end of the file. Below is a summary of the redirection operators:

Standard Input	Standard Output	Standard Error	
<			Redirect Input from file
	>	>&	Redirect Output to file
	>>	>>&	Redirect Output to EOF

Output can be redirected to a storage device, printer, modem, or any other valid GNO or GS/OS device. This provides a very powerful means of communicating directly with these devices from within `gsh`. One quick and dirty example of redirection allows a background version of `gsh` to be run on a terminal connected directly through the modem serial port:

```
[1]% gsh < ttya > ttya >& ttya &
```

Pipelines

In addition to the redirection operators, there is one additional operator which gives control over how input and output are handled. The operator is a **pipeline**, "|". Pipelines allow the standard output of one command to be used as the standard input to another command. This is almost equivalent to running the

first command with its output redirected to a temporary file, then running the second command with its input redirected from the temporary file, then removing the temporary file. Pipelines make useful *filter* processes where the output of one command can be sent to another command which filters the output to whatever parameters you give the second command. As an example, you could display all the filenames with the character "a" in their name:

```
[1]% echo foo > file1; echo abc >> file1; echo aabc >> file1
[2]% echo GNO >> file1; echo standard >> file1; echo oof >> file1
[3]% cat file1
foo
abc
aabc
GNO
standard
oof
[4]% cat file1 | grep 'a'
abc
aabc
standard
```

Pipelines are useful when you wish to view lines of text in a file that contain a phrase, or if you want to connect two programs directly, bypassing intermediate files. It is also possible to connect multiple commands with multiple pipelines.

Pipelines are frequently used for paging output. The **coff** program mentioned above prints the output of an OMF disassembly to the screen but does not pause when a key is pressed. In order to pause the display, the output must be piped through a paging utility. The ORCA shell requires that you wait for the entire command to complete execution before the pipeline is processed. However, GNO/ME executes both commands concurrently which allows the **coff** utility to execute while the paging utility displays the program output. GNO/ME comes with two page utilities, **more** and **less**. Complete descriptions of **coff**, **more**, **less** can be found in the electronic manual using the **man** command.

Background Execution of Commands

A major benefit of GNO/ME is *multitasking*. Multitasking is a means of running multiple applications at once (not literally but very close). On the Apple IIGS, GNO/ME accomplishes pre-emptive multitasking by switching among applications that are running in the background. Any GNO/ME utility can be run in the background. Applications running in the background generally run for the same period of time (GNO/ME switches between applications 20 times a second).

To background a shell utility, place the "&" character at the end of the command-line. The GNO shell displays a unique process ID and job number for each backgrounded command.

It is possible to use the background character "&" to separate commands as with the ";" character. Each command with a following "&" is executed in the background.

Up to 32 processes can executed concurrently under the GNO Kernel.

Warning: When you exit the GNO Shell all processes will be terminated including any you may have running in the background.

Below is a sample session with background tasks:

```

[5] script> ps
  ID  STATE  TT MMID  UID    TIME COMMAND
    1  ready   co 1002 0000   0:45 NullProcess
    2  ready   co 1007 0000   0:05 gsh
 138  running co 1006 0000   0:00 ps
[6] script> cmpl +p script.c keep=script > outputfile &
[1] + 141 Running cmpl +p script.c keep=script &
[7] script> ps
  ID  STATE  TT MMID  UID    TIME COMMAND
    1  ready   co 1002 0000   0:45 NullProcess
    2  ready   co 1007 0000   0:05 gsh
 141  waiting co 1006 0000   0:00 cmpl +p script.c keep=script
 142  ready   co 100B 0000   0:00 5/cc
 143  running co 100D 0000   0:00 ps
[8] script> cmpl +p script.asm keep=script1 > output2 & ps ; ls -s
[2] - 145 Running cmpl +p script.asm keep=script1 &
  ID  STATE  TT MMID  UID    TIME COMMAND
    1  ready   co 1002 0000   0:45 NullProcess
    2  ready   co 1007 0000   0:05 gsh
 141  waiting co 1006 0000   0:00 cmpl +p script.c keep=script
 144  ready   co 100E 0000   0:07 5/linker
 145  ready   co 100D 0000   0:00 cmpl +p script.asm keep=script1
 146  running co 100F 0000   0:00 ps
 147  ready   co 1011 0000   0:00 5/asm65816
    3  barf          1 outputfile    6 script.asm    1 script.root
    1  foobar          19 script        3 script.c     36 script.sym
    1  output2         6 script.a      6 script.mac    1 typescript
[9] script> cp script.asm script2 &
[3] 150 Running cp script.asm script2 &

[2] - Done    cmpl +p script.asm keep=script1 &

[1] + Done    cmpl +p script.c keep=script &

[3] - Done    cp script.asm script2 &

[10] script> ps
  ID  STATE  TT MMID  UID    TIME COMMAND
    1  ready   co 1002 0000   0:45 NullProcess
    2  ready   co 1007 0000   0:05 gsh
 151  running co 1006 0000   0:00 ps

```

The first command line sends the **ps** command to the shell. **ps** lists the processes currently being executed by the GNO kernel. The processes named **gsh** and **NullProcess** are always present. For a complete description of the **ps** command see Chapter 4 **Kernel Commands**.

When a command is executing in the background, keyboard input is not sent to it. However, output is still treated in the same way. If the command sends output to the standard output or standard error, the screen will become cluttered. Try this example:

```

[1]% ls -l&
[2]% ls -l

```

Both the output of commands #1 and #2 will be sent to the screen. After command #1 is entered and you begin typing command #2, you will see the output of the first "**ls -l**" command being sent to the screen while you enter command #2. Utilities which produce output should have their standard output and standard error redirected to a file when they are executed in the background. See Chapter 3 **Redirecting Input and Output**.

Executing commands in the background hinders the performance of the Apple IIGS. This is not too noticeable when one or two commands are being executed but performance will degrade more noticeably as more commands are started. The Apple IIGS was not designed as a multitasking computer so the performance of GNO/ME should be understandable. If you have an accelerator (such as the Transwarp GS or Zip GS) installed, performance of multiple tasks will be acceptable.

Job Control

Now that command backgrounding and multitasking have been discussed, some more definitions can be mentioned. A process is a command which has been submitted to the shell for execution. **Gsh** contains a set of special commands which make dealing with processes much easier. **gsh** treats each command entered at the command-line as a **job**, where a single job may contain multiple processes. For example:

```
% ls                                {one command, one process, one job}
% ls ; ps                           {two commands, two processes, two jobs}
% ls & ps                           {two commands, two processes, two jobs}
% ls | more                          {two processes, one job}
```

When a job is run from the shell, it can be in several modes of operation. Jobs can be in any of three states: **running**, **stopped**, or **done**. A job can be executing in either the foreground or the background.

Commands exist to place a job in any mode of operation. When a job is run directly from a command-line it is running and it is in the foreground. Since the command-line cannot be accessed, two special keys have been defined: **^c** kills the job and **^z** will stop the job. When the job is killed, it is gone forever, but a stopped job can be restarted. When a job is stopped, the kernel suspends each of the processes in the job.

Jobs that are running in the background or have been stopped can be accessed using several built-in commands. The **bg** command will place a job in the background, placing it in the running state if necessary. The **fg** command will similarly place a job in the foreground, and the **stop** command will stop a backgrounded job. The **kill** command will terminate a job.

Each time **job control** is accessed, a special job status line is displayed following the command. The first item on the left in brackets is the job number. Next is a single character, either a '+', '-', or a blank. The '+' designates the currently accessed job, the '-' is the previously accessed job, and all other jobs are not specified. The **jobs** command will display a list of all jobs.

This example was used in Chapter 3 **Background Execution of Commands**, but now more of the notation will be understandable

```
[5] script> ps
  ID  STATE  TT MMID  UID    TIME COMMAND
   1  ready   co 1002 0000   0:45 NullProcess
   2  ready   co 1007 0000   0:05 gsh
 138  running co 1006 0000   0:00 ps
[6] script> cml +p script.c keep=script > outputfile &
[1] +   141 Running cml +p script.c keep=script &
[7] script> ps
  ID  STATE  TT MMID  UID    TIME COMMAND
   1  ready   co 1002 0000   0:45 NullProcess
   2  ready   co 1007 0000   0:05 gsh
 141  waiting co 1006 0000   0:00 cml +p script.c keep=script
 142  ready   co 100B 0000   0:00 5/cc
 143  running co 100D 0000   0:00 ps
[8] script> cml +p script.asm keep=script1 > output2 & ps ; ls -s
[2] -   145 Running cml +p script.asm keep=script1 &
```

```

ID  STATE  TT MMID  UID    TIME COMMAND
  1  ready   co 1002 0000   0:45 NullProcess
  2  ready   co 1007 0000   0:05 gsh
141  waiting co 1006 0000   0:00 cml +p script.c keep=script
144  ready   co 100E 0000   0:07 5/linker
.145 ready   co 100D 0000   0:00 cml +p script.asm keep=script1
146  running co 100F 0000   0:00 ps
147  ready   co 1011 0000   0:00 5/asm65816
  3  barf    1 outputfile 6 script.asm      1 script.root
  1  foobar  19 script    3 script.c       36 script.sym
  1  output2 6 script.a   6 script.mac     1 typescript
[9] script> cp script.asm script2 &
[3]      150 Running cp script.asm script2 &

[2] - Done    cml +p script.asm keep=script1 &

[1] + Done    cml +p script.c keep=script &

[3] - Done    cp script.asm script2 &

[10] script> ps
ID  STATE  TT MMID  UID    TIME COMMAND
  1  ready   co 1002 0000   0:45 NullProcess
  2  ready   co 1007 0000   0:05 gsh
151  running co 1006 0000   0:00 ps

```

Each of the special commands, `bg`, `fg`, `stop` and `kill`, take an argument which specifies the job to perform the operation on. The argument is either a number specifying the process id, or a '%' followed by one of the following: '+' or '%' for the current job, a '-' for the previous job, or a number to specify any specific job. If nothing follows the '%' or the argument is missing, then the current job is the default.

```

[1] gno> cat gshrc

###
#
# GNO 2.0 gshrc file
#
###
#
# Initialize our environment
#
set path=":right:gno:bin :right:gno:usr:bin"
set prompt="[%h] %S%t%s %C> "
set home=":right:gno:user:root"
set term=gnocon
export path prompt home term
setenv history=100 savehist=25
###
#
#Set up standard prefixes for utilities.
#
###
prefix 2 :software:orca:libraries
prefix 3 :software:orca
prefix 4 :software:orca:shell
prefix 5 :software:or^Z
[1] + Stopped (signal) cat gshrc
[2] gno> jobs

```

```

[1] + Running cat gshrc
[3] gno> bg           {output was not redirected, so screen gets cluttered}
[1] + Running cat gshrc
ca:languages
prefix 6 :software:orca:utilities
prefix 7 :tmp
###
#
# Set up prefixes for Orca2.0(tm)'s benefit
#
###
prefix 13 :software:orca:libraries
prefix 14 :software:orca
prefix 15 :software:or
[4] gno> stop           {Incredibly fast typing :-} }
[1] + Stopped (signal) cat gshrc
[5] gno> jobs
[1] + Running cat gshrc
[6] gno> fg
[1] + Running cat gshrc
ca:shell
prefix 16 :software:orca:languages
prefix 17 :software:orca:utilities
alias ls 'ls -CF'
alias dir 'ls -al'
alias cp 'cp -i'
alias rm 'cp -p rm'
alias mv 'cp -p mv'
setenv usr^Z
[1] + Stopped (signal) cat gshrc
[7] gno> jobs
[1] + Running cat gshrc
[8] gno> kill %1
[9] gno> jobs
[10] gno>

```

There is one additional way that a job may be stopped. If the job is placed in the background and it attempts to read from the console, the job will be stopped, and the status line says "(tty input)" as the reason for the job being stopped. The job should be foregrounded so that the user may enter input to the program. It can then be placed back in the background as necessary (with ^Z and bg).

Working with Pathnames

To move easily to directories descended from the home directory, **gsh** provides the "~" (tilde) character. This character represents the home directory. Therefore, if your home directory was ":hard:gno:user:root", you could use the command "**cd ~**" to move to the home directory (note that **cd** without any arguments also defaults to the home directory). To move to subdirectories of the home directory, you could use the command "**cd ~/dir1**" command. The tilde character is recognized by **gsh** before the command is interpreted.

Another special sequence "..", when used as part of a pathname, will strip the last path between pathname separators. For example, the pathname "/dev/gno/.." would be expanded to "/dev". The "/gno" path is stripped as it is before the periods. This provides an excellent way to backup into your directories. "Backing up" is limited by the volume directory of the device being used.

Additionally, the character ".", can be used to signify the current directory.

Examples:

```

[/dev/gno]% cd ~                                { change to home directory }
[/dev/gno]% cd ~/src
[/dev/gno/src]% cp ~/file1 file2                 { copy /dev/gno/file1 to /dev/gno/src/file2 }
[/dev/gno/src]% echo ~/..
/dev/gno/..
[/dev/gno/src]% cd ../../
[/dev]% mkdir ~/utilities                         { create directory /dev/gno/utilities }
[/dev]% mkdir ./libraries                         { create directory /dev/libraries }
[/dev]% cd ~
[/dev/gno]% cp src/file1 .                       { copy /dev/gno/src/file1 to /dev/gno/file1 }

```

Pathname Expansion

Many utilities supplied with **gsh** take, as an argument, a filename or filenames. The shell utilities **cat**, **ls**, **grep**, and **cp** can take multiple filenames as arguments. If you wish to invoke any of these utilities on filenames that have a sequence of characters in common (ie. AND, APPLE, SHK, TXT, FILE2, FILE3, etc.), **gsh** provides special characters, called regular expressions or wildcards, which match multiple filenames without having to enter all filename arguments manually.

*	Matches any string of characters.
?	Matches a single character.
[abc]	Matches any of the characters enclosed in brackets.
[^abc]	Matches any of the characters not enclosed in brackets.
[a-c]	Matches the ascending sequence of characters enclosed in brackets.

This method of matching filenames is known as **globbing**. **gsh** performs globbing on the word prior to executing the command. The following **gsh** session illustrates file globbing:

```

[1]% cd /dev/gno/utilities
[2]% ls
:dev:gno:utilities
CONV          Crunch          CrunchIIGS      DeRez          DiskCheck
DumpObj       Duplicate       EMACS          Equal          Express
Files         LinkIIGS        MakeBin        MakeDirect     OrcaDumpIIGS
Prizm         ResEqual       Search         canon          choose
clrff        cmdfix         coff          compact        count
detab         dir            dirff         dumpfile       eject
emacs.doc     emacs.hlp      emacs.rc       emacs.tut      help
init          join           link          macgen         makelib
mem           online        pageeject     pause          pwd
src
[3]% ls e*
:dev:gno:utilities
EMACS          Equal          Express         eject          emacs.doc
emacs.hlp      emacs.rc       emacs.tut
[4]% echo *r *m
dir Prizm mem
[5]% echo *i*
cmdfix CrunchIIGS Prizm DiskCheck Duplicate Files init join LinkIIGS makelib
MakeBin MakeDirect link dirff dumpfile online OrcaDumpIIGS dir
[6]% echo NoMatch*
No match.
[7]% echo [a-f]*

```



```

coff canon cmdfix compact Crunch CrunchIIGS DeRez DiskCheck DumpObj Duplicate
EMACS emacs.doc emacs.hlp emacs.rc emacs.tut Equal Express Files choose clrff
count detab CONV dirff dumpfile eject dir
[8]% echo [a-fs-t]*
coff canon cmdfix compact Crunch CrunchIIGS DeRez DiskCheck DumpObj Duplicate
EMACS emacs.doc emacs.hlp emacs.rc emacs.tut Equal Express Files choose clrff
count detab Search src CONV dirff dumpfile eject dir
[9]% echo emacs?*
EMACS emacs.doc emacs.hlp emacs.rc emacs.tut
[10]% echo [^a-f]*
Prizm help init join LinkIIGS makelib MakeBin MakeDirect link mem ResEqual
Search src online pageeject pause OrcaDumpIIGS pwd macgen
[11]% echo [^a-fs-t]*
Prizm help init join LinkIIGS makelib MakeBin MakeDirect link mem ResEqual
online pageeject pause OrcaDumpIIGS pwd macgen
[12]% echo ???
mem src pwd dir
[13]% echo ?
No match.
[14]% echo "???"
???
[15]% do you have a light?
No match.

```

As can be seen by the above example, character matches are case insensitive. The ProDOS file system treats the filenames "file" and "FILE" as the same file. **gsh** recognizes this and does not detract from the underlying file system.

File globbing makes passing arguments to commands much easier and much more powerful. You could easily use "*.c" as an argument in a number of ways:

```

[1]% ls *.C
...
[2]% cc *.C
...
[3]% more *.C
...

```

{ lists all filenames ending in ".C" }

{ compiles all files ending in ".C" }

{ displays contents of all files ending in ".C" }

Quoting Special Characters

Beginning with Apple IIGS System Software 6.0, GS/OS is able to read files from Macintosh computers. The Macintosh uses a filesystem known as HFS, which allows filenames to contain any character except the colon (":"). Because a filename such as "emacs?*" is valid under HFS, care must be taken or unexpected results will occur. The word "emacs?*" was used as a regular expression above to specify a list of filenames beginning with the word "emacs" and one or more trailing characters. **gsh** does provide a way to pass an argument which contains special shell characters to a command. This is known as quoting an argument. There are three different ways to quote an expression:

The single quote will quote everything between the single quote marks. Thus, to display the contents of a file on an HFS volume named "emacs?*":

```
% more 'emacs?*'

```

The double quote will quote everything between the double quote marks except variables. See Chapter 5 for more on variables.

```
% echo "emacs?* $home"
emacs?* /dev/gno
```

The backslash is used to quote one character. To pass "emacs?*" as a regular expression using the backslash, enter the following:

```
% ls emacs\?\*
```

One additional purpose of the quoting mechanism built into **gsh** is to add spaces to command arguments. Each command and its arguments is separated by a space. Multiple spaces between arguments are treated as one space. Thus, consider the following:

```
% echo a      b c
a b c
% echo 'a      b c'
a      b c
```

How gsh Finds a Command

gsh has a special variable, **PATH**, which specifies the directories and order of directories to search for shell utilities. This variable is often setup in the *gshrc* file although it can be changed as often as needed. The purpose of the **PATH** variable was discussed in Chapter 1 **Customizing the Shell Environment**.

When **gsh** starts up, it searches all directories specified in the **PATH** variable and establishes a table of all commands, called a hash table. Because of this table, **gsh** "knows" where a command is and can execute the command much faster than searching through all directories every time the command is entered.

The search process begins with alias names. See Chapter 3 **Using aliases as shorthand**. If an alias is found that matches the command, the alias is replaced with its value and the command-line is again parsed. If it was not an alias, **gsh** checks to see if it was a special built-in utility. The search process then searches for the name in the hash table. If an entry is found in the hash table, the path name of the command is retrieved and the command is executed. If an entry is not found, the current path is searched. If the command name is not found, an error results.

When the **PATH** variable is changed, **gsh** does not automatically recreate the command hash table. You need to issue the command **rehash** to recreate the hash table. The more pathnames specified, the greater the delay in starting **gsh** and in invoking the **rehash** command. The following shell script changes **PATH** and invokes the **rehash** command in one step.

```
echo Resetting PATH variable $PATH to $1
set path=$1
rehash
```

The **\$1** variable will be expanded with the first argument passed to the script.

rehash should also be used if a new utility is copied to one of the directories specified in the **PATH** variable. Of course, it is possible to specify the absolute pathname of any command, but this is undesirable if the command is frequently used.

Chapter 4

Built-in Commands

Built-ins vs EXE Commands

The term "built-ins" is used to describe commands that exist within the shell itself. These utilities run faster than external commands because the code is already loaded into memory. Files of type "EXE", on the other hand, must be loaded into memory by **gsh** and executed. If an EXE command is executed again, it might, again, have to be loaded into memory. This results in longer execution time for the program.

gsh has a number of built-in commands which allow you to work with the shell, the GNO kernel, and the shell environment.

The following section describes the commands that are built-in to **gsh**. The "[..]" character sequence represents an optional argument to a command. The term "SIGNAL" is used to represent one of the signal names or numbers listed in Appendix D **Signals**. The sequence "..." means the command accepts multiple arguments of the same type as the argument before the "..." sequence. The sequence "{..}" is used to represent a set, which is a list of possible arguments to choose from.

Shell Commands

Shell built-ins provide support for external shell commands (i.e. EXE files and shell scripts) and provide some commands used in every-day work.

bindkey [-l] function string

Bindkey is used to customize the shell's command-line editor. Any key on the keyboard can be mapped to any of a number of functions. The various functions are as follows:

backward-char	move cursor left
backward-delete-char	delete character to left
backward-word	move cursor left one word
beginning-of-line	move cursor to beginning of line
clear-screen	clear screen and redraw prompt
complete-word	perform filename completion
delete-char	delete character under cursor
down-history	replace command line with next history
end-of-line	move cursor to end of line
forward-char	move cursor to the right
forward-word	move cursor one word to the right
kill-end-of-line	delete line from cursor to end of line
kill-whole-line	delete the entire command line
list-choices	list file completion matches
newline	finished editing, accept command line
raw-char	character as-is
redisplay	redisplay the command line
toggle-cursor	toggle between insert and overwrite cursor
undefined-char	this key does nothing
up-history	replace command line with previous history

Keys are bound to functions, not vice-versa. This means that you can have any number of commands refer to the same function. For example, the default bindings have `CTRL-A` and `OA-<` both bound to `beginning-of-line`.

Most of the function names are self-explanatory, and are explained in Chapter 2, but a few deserve discussion. `raw-char` is what you should bind a key that should be inserted into the command-line as-is. The regular printable ASCII set, such as the letters a-z, numbers, etc. are bound to `raw-char`. Control characters should not be bound to `raw-char` because the command-line editor will become confused (most control characters act as special GNO/ME console feature codes - see the *GNO Kernel Reference Manual*, Chapter 4 **TextTools Replacement**).

Any keystroke that should be rejected by the editor should be bound to `undefined-char`. By default, this includes control characters and OA-sequences that are not assigned to any editing features. Any key bound to `undefined-char` will cause `gsh` to beep and ignore the key.

You can actually bind key sequences, not just keystrokes, to functions. There is no limit other than memory to how many characters are in a command sequence.

Because terminals do not have the OA (Open Apple) key, OA is actually mapped by the kernel to a two-character sequence consisting of `ESC` and the key. For example, `OA-Y` would actually produce `ESC-Y`.

Control characters in the `string` are represented in `^X` format; e.g. `CTRL-A` is represented by `^A`. `ESC` (and `OA`) is represented by `^[`.

Examples:

```
% bindkey kill-end-of-line ^K    map CTRL-K to kill-end-of-line (like Emacs)
% bindkey clear-screen  ^[[X      map OA-CLEAR to clear-screen
```

commands

Displays a list of all built-in shell commands.

```
cd [pathname]
chdir [pathname]
```

Changes the current working directory to `pathname`. If `pathname` is not given, the default `HOME` directory (i.e. the value of the `HOME` environment variable) is used. This makes it easy to move back to your home directory. Under `gsh`, unlike most UNIX shells, the `cd` is not necessary, except to change automatically to your `HOME` directory. If a command is not a built-in or EXE file, but is instead the name of a directory, a `cd` is implied and performed on the directory unless the `NODIREXEC` variable has been set.

```
[1] gno> cd utilities
[2] gno/utilities> echo $HOME
/dev/gno
[3] gno/utilities> cd
[4] gno> utilities
[5] gno/utilities> ../utilities
[6] gno/utilities> ~
[7] gno> _
```

clear

This command takes no arguments. When invoked, the screen will be cleared.

df

This command takes no arguments. When invoked, a listing of free blocks for every block device is given. In addition, the device name, type, file system, and capacity is listed.

```
[1] gno> df
```

.d##	Volume	Device	Free	Total	Capacity	System
.d1	:Procyon	.CVTECH.S6.A	3166	41037	92%	ProDOS
.d2	Apple 3.5 Drive	.APPLEDISK3.5A				
.d3	Apple 3.5 Drive	.APPLEDISK3.5B				
.d4	Console Driver	.CONSOLE				
.d5	:Day	.CVTECH.S6.B	15881	65535	75%	ProDOS
.d6	:Night	.CVTECH.S6.C	39274	65535	40%	HFS
.d7	:Left	.CVTECH.S6.D	5365	51776	89%	ProDOS
.d8	:Right	.CVTECH.S6.E	27477	65535	58%	ProDOS
.d9	:Software	.CVTECH.S6.F	3289	40960	91%	ProDOS
.d10	:RAM5	.RAMDISK	505	512	1%	ProDOS
.d11	:Conner 40	.AFP1	7368	80604	90%	AppleShare
.d12	AppleTalk fsd	.AFP2				
....						
.d24	AppleTalk fsd	.AFP14				
.d25	AppleTalk RPM	.RPM				
.d26	AppleTalk Main	.APPLETALK				
.d27	Serial Modem	.SERIAL2				
.d28	:system	.APPLESCSI.HD01.	27561	50773	45%	ProDOS
.d29	:dev	.APPLESCSI.HD01.	43431	65108	33%	ProDOS
.d30	:usr	.APPLESCSI.HD01.	44376	48725	8%	ProDOS

Volume - Name of device.

Device - GS/OS device name.

Free - Total number of free blocks on device.

Total - Total number of blocks on device.

Capacity - Percentage of used blocks on device.

System - Format of file system. With System Software 6.0.1, the file systems available are ProDOS, HFS, Pascal, MS-DOS, and DOS 3.3, and some CD-ROM formats.

echo [-n] arg [arg ...]

Expands the "arg" expression(s) and outputs them to the screen. If the "-n" switch is specified, a NEWLINE character is not output after the last "arg" expression. Special escape sequences may also be included in the arguments, similar to those used in C strings:

\b - Backspace

\f - Clears screen (form feed)

\n - Newline

\r - Return

\t - Tab

\nnn - A decimal ASCII code. nnn represents the ASCII code.

```
[1] gno> echo Hello World
```

```
Hello World
```

```
[2] gno> echo -n Hello World
```

```
Hello World[3] gno> echo $PATH $HOME 'come get to $gnome'
```

```
/dev/gno/utilities /dev/orca/utilities /dev/gno come get to $gnome
```

exit

Exits the shell or terminates a shell script.

history

This command displays the list of previous command-line entries. The number of entries saved is set in the \$HISTORY variable.

pushd
popd [+n]
dirs

These three commands maintain the shell's *directory stack*. Let's say you're working in a directory `/src/myprogs/class/program.1/`, and you want to temporarily go to another directory. Instead of having to 'cd' there and 'cd' back to a very long directory name (i.e., lots of typing), you can use the `pushd` command, like so:

```
gsh> pushd /etc                                { start in /src/myprogs/class/program.1/ }
...
gsh> popd                                       { back in /src/myprogs/class/program.1/ }
```

The `pushd` command stores the current directory on a stack, and then changes the current directory to the argument. When you want to go back to the original directory, type `popd`. The shell will pull the last directory off the stack and 'cd' to that directory.

The `popd` command when given an argument of `+n` will remove the `n`'th directory from the stack. It does not change to that directory.

The `dirs` command displays the current directory stack.

pwd

Displays the current working directory. This is useful if you have not configured the `$PROMPT` string to output your current working directory.

```
[1]> cd $HOME
[2]> pwd
/user/root
[3]> cd utilities
[4]> pwd
/user/root/utilities
```

source

When a script is executed, `gsh` creates a new process to run the script. As a result, scripts cannot change the shell's prefixes and certain other parameters. Instead of executing the script directly, you may use the `source` command which does not create a new process to execute the script. Thus, the `source` command is effectively exactly like typing all the commands in the script from the keyboard.

tset

The `tset` command causes the shell to reread the `/etc/termcap` file and reset its output system to use the terminal type specified in the `$TERM` variable. On startup, after reading the `gshrc` file, `gsh` automatically does a `tset`. `gsh` also automatically does a `tset` whenever the `$TERM` variable is changed with the `set` command. You would use `tset` manually if, for example, a utility changed the value of `$TERM`.

which command_name [command_name ...]

Let's say that you are working on a new version of the venerable shell utility `ls`. Since a search of the hash table is done before searching the current directory, you might accidentally be using the wrong version of the command. You make changes and run the new program, but your changes don't seem to appear! Use the `which` command to check your sanity. Which also comes in handy in locating duplicate program names in the `$PATH` directories (for example, an `ls` in both `/bin` and `/usr/bin`.)

The way to access a utility in the current directory which has the same name as a program in the `$PATH` is to prefix the command name with `./`, as in `./ls`. Also, see `unhash`.

Kernel Commands

gsh provides a set of commands to control the GNO kernel. These commands mainly deal with job control. See Chapter 3 **Job Control**.

bg { %job | pid }

Starts the specified job, if stopped, and places it in the background.

fg { %job | pid }

Starts the specified job, if stopped, and brings it into the foreground.

jobs [-l]

Displays a list of the shell's jobs. If the "-l" switch is specified, the process id is included in the job list.

```
[1] gno> cmpl foo.c keep=foo &
[1] + Running cmpl foo.c keep=foo &
[2] gno> echo hello
hello
[3] gno> jobs
[1] + Running cmpl foo.c keep=foo &
```

kill {[-SIGNAL] %job | pid | [-l]}

The kill command will send the signal SIGNAL to the process number pid. The ps command documented below describes how to list all process ID's currently executing.

SIGNAL can be either a numeric value or string representing the signal to be sent to the process. All signals are documented in Appendix D **Signals**, with numeric and string value listed. Alternatively, specifying the -l option will list all the signals and their names.

If the process number isn't known, but the job number is, replace the pid with a '%' followed by the job number.

ps

This command takes no arguments. When invoked, a list of all currently running processes is displayed.

```
[2] 9:52pm root> ls -lR :hard:gno > /ram5/dev &
[1] + 35 Running ls -lR :hard:gno &
[3] 9:53pm root> ps
  ID  STATE  TT MMID  UID    TIME COMMAND
   1  ready   co 1002 0000   0:26 NullProcess
   2  ready   co 1005 0000   0:02 gsh
  35  ready   co 100A 0000   0:01 ls -lR :hard:gno
  36  running co 1007 0000   0:00 ps
[4] 9:53pm root>
[1] + Done    ls -lR :hard:gno
```

ID - A unique process ID assigned to a command by GNO. Use this number to reference any process.

STATE - Current state of the process. Each process can be in any of the following states:

RUNNING	the process is currently in execution.
READY	the process is not currently executing, but is ready to be executed as soon as it is assigned a time slice.
BLOCKED	the process is waiting for a slow I/O operation to complete (for instance, a read from a TTY).
NEW	the process has been created, but has not executed yet.

SUSPENDED	the process was stopped with SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU.
WAITING	the process is waiting on a semaphore 'signal' operation. Programs waiting for data from a pipe have this state.
WAITSIGCH	the process is waiting to receive a SIGCHLD signal.
PAUSED	the process is waiting for any signal.

TTY - Terminal connected to process.

MMID - Memory Manager ID assigned to process.

UID - ID of the user who initiated the process.

TIME - How much CPU time this process has used. This is not the elapsed time of the process.

COMMAND - Command-line string used to invoke process.

setdebug (val | (+|-)flag)

Turns GNO kernel debugging code on or off. The value passed consists of a bit field, where each bit specifies a different type of debugging code to activate. An alternate method is to provide a list of debug flags, either preceded by a '+' or a '-'. Those preceded by a '+' are activated, and those preceded with a '-' are deactivated. All debugging is deactivated by passing a value of 0. Running setdebug with no arguments returns a list of the debugging flags. Legal flags include:

gsotrace	- Trace GS/OS calls
gsoblocks	- Trace GS/OS parameter blocks
gsoserrors	- Trace GS/OS errors
pathtrace	- Trace GS/OS pathnames
sigtrace	- Trace signals
sysrtrace	- Trace system calls

stop (%job | pid)

Stops the execution of all processes in a specified job.

Environment Commands

The last set of commands, environment commands, modify the **gsh** environment. Many of these commands have been used in other parts of this manual and, therefore, should not be new.

alias [name] [value]

Creates an alias for a string. When this alias is referenced as a command, value will be expanded into the command line. For commands that require many arguments or have several steps, you could set up an alias to save typing. You can also use aliases to create new names for commands. To obtain a list of all aliases, invoke **alias** with no arguments. To list the value of an alias, invoke **alias** with name only. Here are some alias examples:

<u>Alias</u>	<u>Command Name</u>
ll or list_long	ls -l
backup_sys	cp -r /system /BackupDrive&
backup_home	cp -r \$HOME /BackupDrive&
print	echo
catalog	ls -l
delete	rm
copy	cp
type	more
rename	mv

export [variable ...]

When a shell environment variable is marked as exportable, any process that is created from within the current process (most likely **gsh**), will be passed copies of the exported variables. See **setenv** and Chapter 5 **Scope of shell variables**.

hash

Displays a list of all commands currently in the shell's hash table; i.e., a list of commands in the various \$PATH directories.

prefix [prefixnum [prefixname]]

GNO maintains a list of 32 'prefixes' for each process. Prefixes allow the user to reference a directory with a number. While **gsh** provides this ability with environment variables, the prefix command exists to support the ORCA compilers and other utilities that are dependent on certain GS/OS prefixes. Appendix B contains a list of these prefixes and their "default" meanings, as documented in the *"Apple IIGS GS/OS Reference"*, Volume 1.

If prefixname is not given, the value of prefixnum is displayed. If neither argument is given, a list of currently assigned prefixes is displayed.

rehash

To decrease the time spent searching for a command, **gsh** builds a table of all commands which were found in the pathnames set in the \$PATH variable. When a command is invoked, only this list is searched. When the \$PATH variable is changed, **gsh** must rebuild this list. The **rehash** command tells **gsh** to rebuild the list.

While the old list is still active, if \$PATH is changed and one of the previous search paths is no longer online, **gsh** will try and execute the command from the offline device, resulting in a command failure.

To make this a one-step process, the `change.path` shell script listed in Chapter 5, **Accessing shell variables** can be used.

```
set [var] [value] [(var value) ...]
set var=value [var=value ...]
setenv [var value]...
```

Use these command to create or modify environment variables. If **set** is invoked with no arguments, a list of the current environment variables is displayed. If only `var` is given as an argument, the value of `var` is displayed. To set or reset a variable, use both the `var` and `value` arguments. There are two ways to set a variable, either by "`var value`" or "`var=value`". To set multiple variables at once, simply list them all on the command line as shown above.

setenv works just like **set**, but automatically exports the variable(s) or lists only exported variables.

When using **set** or **setenv** to view a list of variables, exported variables appear in ALL CAPS.

unalias name [name ...]

To remove an alias from the alias list, use this command. To remove multiple aliases with one command, specify all the aliases on the command line.

unhash

To disable the internal hash table created with the **rehash** command, use this command. This is useful if you wish to use only utilities in the current working directory (during testing, for example).

unset var [var...]

To remove a variable from the environment, use `unset`. `unset` accepts multiple names if more than one variable is to be deleted. Future attempts to access the variable `var` will result in an error or a NULL string, depending on the circumstances.

```
[8] 9:57pm root> set
PAGER = less
PATH = :hard:gno:bin :right:gno:usr:bin
user1 = foo
USRMAN = /usr/man
HISTORY = 30
status = 227
figignore = .a .root .sym
TERM = gnocon
LESS = -e
PROMPT = [%h] %S%t%s %C>
HOME = :hard:gno:user:root
user = user1
nonewline = 1
[9] 9:57pm root> unset user1
[10] 9:57pm root> unset user
[11] 9:57pm root> set
PAGER = less
PATH = :hard:gno:bin :right:gno:usr:bin
USRMAN = /usr/man
HISTORY = 30
status = 0
figignore = .a .root .sym
TERM = gnocon
LESS = -e
PROMPT = [%h] %S%t%s %C>
HOME = :hard:gno:user:root
nonewline = 1
```

Chapter 5

Shell Variables

And then one day, hurray! Another way for gnomes to say hooray!
Syd Barret, The Gnome

Using shell variables

gsh supports **variables** in the shell environment. These variables can be used by any shell utility or script. Many EXE files and shell scripts pre-define certain shell variables that contain formatting options or other options for a specific utility. As an example, the **ls** utility looks for the variable **TERM** that defines the terminal type currently being used. When **ls** is started, it reads the value of the **TERM** variable and avoids printing Apple II specific MouseText characters if the set terminal does not support them.

gsh has set aside certain variables for its specific use. Shell utilities should be aware of these variables and use them appropriately. Use caution when changing shell variables, because the change could affect more than just the shell.

Scope of Shell Variables

There are two types of processes that are involved in any discussion of a multitasking system. The original process, **gsh** for example, is called a parent process. If **gsh** invokes a process, such as **ls**, **cp**, or **mv**, that process is called a child process. It is possible for any process to define a variable. These variables will not be made available to other processes unless the program that defined the variable specifically makes them available.

The *export* command makes variables defined by one process available to its children. See the example *gshrc* shell script shown in Chapter 1 **Customizing the Shell Environment**. In the case of the shell, most of its variables are exported and, therefore, all shell utilities can read the value of a shell variable. However, programs cannot change the value of a shell variable. In general, executables share their environment with that of the shell, so that a utility can change variables in its parent's environment. This allows communication between programs and the shell.

Description of Pre-defined Shell Variables

The following variables have special meaning to **gsh**. Shell variables are not case sensitive.

- `$0, $1, $2, ...`
String values that contain the arguments to a shell script. Variable 0 contains the name of the script. The first argument begins with variable 1 and so on. To expand a variable, use the dollar character, "\$" (See Chapter 5 **Accessing shell variables**)
- `$<`
When encountered, the variable is expanded using a value obtained from standard input. This provides a means of obtaining user input in script files. Note that the shell variables are expanded before the command-line is executed (See Chapter 5 **Accessing shell variables**) When prompting the user for input, be sure that the prompt is in a separate command-line than the `$<`. Also, if the user wishes to enter a value with spaces, he must quote what he types with double-quotes.

\$ECHO	A boolean value that, if defined, will cause commands in a shell script to be echoed to standard output.
\$FIGNORE	This variable, if set, contains a list of filename suffixes. When doing command or filename completion, gsh will ignore any filename with a suffix listed in FIGNORE . For example, you might want to set <code>fignore=".A .ROOT .SYM"</code> to ignore object files and other compiler droppings.
\$HISTORY	A numeric value that contains the number of history commands (command-lines) remembered. If the value is 0 or HISTORY is undefined, all commands will be remembered. Previous command-lines can be called back with the UP-ARROW and DOWN-ARROW (See Chapter 2, History Editing Commands).
\$HOME	The HOME directory is the main directory of the shell; it is the directory gsh defaults to when it starts. The tilde character ("~") can be used as a shorthand method of accessing the HOME directory (as discussed in Chapter 3, Tilde Expansion).
\$IGNOREEOF	A boolean value that, if enabled, will prevent ^D from exiting the shell.
\$NOBEEP	A boolean value that, if set, will prevent gsh from sounding the speaker when errors occur while editing a command-line.
\$NODIREXEC	A boolean value that, if set, will disable gsh 's feature of treating directory names as commands; i.e. if a directory is specified as a command, gsh will move to that directory as though the <code>cd</code> command was being used.
\$NOGLOB	A boolean value that, if enabled, will disable filename globbing. Command arguments will be passed to their commands "as-is", without any wildcard expansion.
\$NONEWLINE	A boolean value that, if set, will disable carriage returns being output before and after command execution. Previous examples given in this manual have this option set.
\$PATH	A string value that defines the pathnames where shell scripts, EXE utilities, and SYS16 programs can be found (See Chapter 3, How gsh finds a command). Because GS/OS uses colons as separators in pathnames, gsh cannot allow colons to be used as separators in the PATH variable, as UNIX does. If one of the path entries has a space within it (which is possible with the HFS FST), then the space should be quoted with a backslash, "\".
\$PRECMD	Actually an <u>alias</u> and not a variable, if PRECMD is defined it will be executed just before gsh prints the prompt for a command line. For example, <code>alias precmd qtime</code> will print the time in English text before every prompt.
\$PROMPT	When gsh prompts you to enter a command, the prompt that appears before the cursor can be customized for your gsh environment. If PROMPT is undefined, the default prompt of "% " is used. The prompt string recognizes certain character sequences in the PROMPT variable and interprets them accordingly. The following are the special characters: <div style="margin-left: 20px;"> %h, %!, ! Current history number. %t, %@ Current time of day in 12 hour am/pm format. %d, %/ Current working directory. %~ Current working directory with tilde ("~") replacement. %c, %C, % . Trailing component of current working directory. %S, %s Inverse mode ON (%S) / OFF (%s). </div>

<code>%U, %u</code>	Begin and end Underline mode (only on terminals that support underline. <code>gnocron</code> will use inverse mode instead.)
<code>%%</code>	Displays a single "%" character.
<code>%n</code>	User name (contents of <code>\$USER</code> variable).
<code>%W</code>	The date in mm/dd/yy format.
<code>%D</code>	The date in yy-mm-dd format.
<code>\n</code>	Newline
<code>\r</code>	Carriage Return
<code>\t</code>	Tab
<code>\b</code>	Bell (beep)

Examples of prompts already used in this manual are "[1]%", and "[/user/root]%" These were created by using the commands **set prompt "[%h]%"** and **set prompt "[%d]%"**.

`$PUSHDSILENT`

If this variable is defined, **gsh** will not print the directory stack after any of the directory stack commands (See Chapter 4 **pushd**, **popd**, .)

`$SAVEHIST`

A numeric value that contains the number of commands to save to disk when exiting **gsh**. These commands are then read back in when **gsh** is restarted which allows old commands to be reused. If the value is 0 or `SAVEHIST` is undefined, no commands will be saved to disk.

`$TERM`

This variable contains the name of the terminal emulation that the shell and other applications should use. By default, it is 'gnocron'. When the shell encounters a 'set term' command, it automatically calls the **tset** command to reload the termcap information. See Appendix F **Termcaps**.

`$TERMCAP`

This variable specifies the location of the *termcap* file. The shell and other applications look for *termcap* in the /etc directory, but if `TERMCAP` is set, the fully specified termcap file is used instead. This allows users to install custom termcap entries. See Appendix F **Termcaps**.

`$USER`

A string that represents the name of the current user. This variable is usually set by **login(1)**.

Accessing shell variables

Shell variables are defined or changed with the **set** command. The **unset** command is used to delete a variable. See Chapter 4 **Environment Commands** for more information on the **set** and **unset** commands

To access shell variables from the command line or a shell script, use the character "\$" followed by the variable name. The dollar sign character will expand the variable to its value. If you wish to use the dollar sign character in a string from the command line, remember to enclose it in single quotes or use the "\" escape character. If you use double quotes, the shell will try to expand the variable. To differentiate the variable name from characters that may immediately follow it, the variable name may be optionally surrounded with braces, "{" and "}". This provides a very powerful way of user interaction with shell scripts. As an example,

```
[1] 9:41pm root> set
PAGER = less
SAVEHIST = 25
PATH = :hard:gno:bin :right:gno:usr:bin
USRMAN = /usr/man
HISTORY = 100
status = 0
figignore = .a .root .sym
TERM = gnocon
LESS = -e
PROMPT = [%h] %S%t%s %C>
HOME = :hard:gno:user:root
user = root
nonewline = 1
[2] 9:41pm root> set history 30 user user1 ; set
PAGER = less
SAVEHIST = 25
PATH = :hard:gno:bin :right:gno:usr:bin
USRMAN = /usr/man
HISTORY = 30
status = 0
figignore = .a .root .sym
TERM = gnocon
LESS = -e
PROMPT = [%h] %S%t%s %C>
HOME = :hard:gno:user:root
user = user1
nonewline = 1
[3] 9:41pm root> unset savehist ; set
PAGER = less
PATH = :hard:gno:bin :hard:gno:usr:bin
USRMAN = /usr/man
HISTORY = 30
status = 0
figignore = .a .root .sym
TERM = gnocon
LESS = -e
PROMPT = [%h] %S%t%s %c>
HOME = :hard:gno:user:root
user = user1
nonewline = 1
[6] 9:43pm root> echo 'Current value of $path = ' $path
Current value of $path = :right:gno:bin :right:gno:usr:bin
[7] 9:44pm root> echo "value of $path = " $path
value of :right:gno:bin :right:gno:usr:bin = :right:gno:bin
:right:gno:usr:bin
[9] 9:45pm root> set user1 foo
[10] 9:45pm root> echo $user1
foo
[11] 9:45pm root> echo ${user}1
user11
[12] 9:46pm root> echo "echo '\$path = ' \$path" >> change.path
[13] 9:46pm root> echo "set path \$1" >> change.path
[14] 9:47pm root> echo 'rehash' >> change.path
[15] 9:47pm root> cat change.path
echo '$path = ' $path
set path $1
rehash
[16] 9:48pm root> change.path '/dev/gno/utilities'
```

```
$path = /dev/gno/orca /dev/gno/utilities
hashed 22 files.
[17] 9:49pm root> change.path $PATH
$path = /dev/gno/orca /dev/gno/utilities
hashed 55 files.
[18] 9:50pm root> echo this is $homeplate
this is { variable homeplate wasn't found }
[19] 9:51pm root> echo this is ${home}plate
this is /dev/gnoplate { variable home is found }
```


Appendix A

Sample gsh session

The following represents a sample session in gsh. Many of the features discussed in this manual are reviewed here. Comments are enclosed in braces ("{}").

```
[1] gno> set
savehist = 25
path = /dev/orca/utilities /dev/gno/orca /dev/gno/utilities
history = 25
prompt = [%h] %C>
user = achin
home = /dev/gno
nonewline = 1
[2] gno> echo environment variable PROMPT = $PROMPT
environment variable PROMPT = [%h] %C>
[3] gno> cat gshrc
```

{ gshrc is the startup file. }

```
###
#
# GNO 2.0 gshrc file
#
###
#
# Initialize our environment
#
set path=":hard:gno:bin :right:gno:usr:bin"
set prompt="[%h] %S%t%s %C> "
set home=":hard:gno:user:root"
set term=gnocon
export path prompt home term
setenv history=100 savehist=25
###
#
#Set up standard prefixes for utilities.
#
###
prefix 2 :software:orca:libraries
prefix 3 :software:orca
prefix 4 :software:orca:shell
prefix 5 :software:orca:languages
prefix 6 :software:orca:utilities
prefix 7 :tmp
###
#
# Set up prefixes for Orca2.0(tm)'s benefit
#
###
prefix 13 :software:orca:libraries
prefix 14 :software:orca
prefix 15 :software:orca:shell
prefix 16 :software:orca:languages
prefix 17 :software:orca:utilities
alias ls 'ls -CF'
alias dir 'ls -al'
```

```

alias cp 'cp -i'
alias rm 'cp -p rm'
alias mv 'cp -p mv'
setenv usrman='/usr/man'
set fignore='.a .root .sym'
alias zcat 'compress -cd'
setenv pager=less
setenv less=-e
set nonewline=1
#
# Move to home directory
#
cd
[4] gno> cat gshrc > .ttyb&
                                                    { .ttyb is the printer connected to the printer port. }

[2]
[5] gno> echo File \'gshrc\' sent to printer.
File 'gshrc' sent to printer.
[6] gno> alias l 'ls -F'; l
                                                    { The ls "-F" option displays executable and directory files differently. }

:dev:gno
doc/          foo*          gsh*          gshrc*        kern*
man/          message      message2      orca/         press.release
release.notes samples/      todo         updates       utilities/
[7] gno> history
 1: set
 2: echo environment variable PROMPT = $PROMPT
 3: cat .gshrc
 4: cat .gshrc > .printer&
 5: echo File \'gshrc\' sent to printer.
 6: ls
 7: history
[8] gno> no_file
no_file: Command not found.
[9] gno> message
message: Not executable.
[10] gno> unset USER
                                                    { Environment variables are not case sensitive. }

[11] gno> set PATH

path = /dev/orca/utilities /dev/gno/orca /dev/gno/utilities
[12] gno> set PROMPT

prompt = [%h] %C>
[13] gno> unset PROMPT
% cp message message2&
[2]
% rm doc&
rm: :dev:gno:doc directory
% rm -r doc&
[3]
% ls *
:dev:gno
foo*          gsh*          gshrc*        kern*         man/
message      message2      orca/         press.release release.notes
samples/     todo         updates       utilities/
% echo *
gshrc foo message Finder.Data utilities press.release man release.notes todo
updates gsh kern samples doc orca message2

```

```
% echo g* m*
gshrc gsh message man message2
% echo g?
No match.
% pwd
/dev/gno
% clear ; df
```

.d##	Volume	Device	Free	Total	Capacity	System
.d1	:system	.APPLESCSI.HD01.	25746	50773	49%	ProDOS
.d2	:dev	.APPLESCSI.HD01.	44868	65108	31%	ProDOS
.d3	:usr	.APPLESCSI.HD01.	44526	48725	8%	ProDOS
.d4	:ram5	.DEV2	493	512	3%	ProDOS

```
% which ls cp echo ps
ls: /dev/gno/utilities/ls
cp: /dev/gno/utilities/cp
echo: Shell Built-in Command
ps: Shell Built-in Command
% echo -n foo
foo % 1 .dev2 .applescsi.hd01.01
{ Because of the previous "echo -n", the PROMPT is displayed on the line where echo output ended. }

:ram5
df echo history ls ls.ram5
ls.system prefix ps set which

:system
AppleworksGS/ BASIC.System* HyperCard/ ProDOS* a2fx/
basic.launcher* emacs.icon fontasm/ gutenber/ icons/
kermit/ publish.it/ review shr.pic/ shrinkit/
snowterm/ system/
% alias print 'echo'; alias copy 'cp'; alias delete 'rm'; alias
catalog 'ls -l'; alias rename 'mv'
{ Useful aliases for Orca fans. }

% alias
l: ls -F
print: echo
copy: cp
delete: rm
catalog: ls -l
rename: mv
% set PROMPT='[!] %C> '
[30] gno> cd samples
[31] samples> cml ps.c keep=ps > .dev2/ps&
[2]
[32] samples> cml kill.c keep=kill > .dev2/kill&
[3]
[34] samples> ls dummy_dir
[35] samples> ls -9
ls: unrecognized option '-9'
usage: ls [-acdfgilqrstulACLFR] [name ...]
[36] samples> ls -9 >& .dev2/error
{ Each pathname has a device name entry (cf. ps). }
[37] samples> cat < /ram5/error
ls: unrecognized option '-9'
usage: ls [-acdfgilqrstulACLFR] [name ...]
[38] samples> ls *.c;grep \#\i\n\c\l\u\d\e [^sfd]*.c
{ grep searches for a regular expression in a file and outputs the lines that match. }

comp.c dp.c edit.c ffdi.c gar.c
kill.c link.c ps.c setdebug.c
```

```

kill.c:#include <stdio.h>
kill.c:#include <memory.h>
kill.c:#include "3/work/gno/conf.h"
kill.c:#include "3/work/gno/proc.h"
kill.c:#include "3/work/gno/kerntool.h"
gar.c:#include <stdio.h>
gar.c:#include <orca.h>
gar.c:#include "3/work/gno/kerntool.h"
gar.c:#include <gno/signal.h>
ps.c:#include <stdio.h>
ps.c:#include <memory.h>
ps.c:#include <string.h>
ps.c:#include <orca.h>
ps.c:#include "3/work/gno/conf.h"
ps.c:#include "3/work/gno/proc.h"
ps.c:#include "3/work/gno/kvm.h"
edit.c:#include <stdio.h>
edit.c:#include <stdlib.h>
edit.c:#include <string.h>
edit.c:#include <ctype.h>
edit.c:#include <types.h>
edit.c:#include <gsos.h>
edit.c:#include <shell.h>
edit.c:#include <orca.h>
edit.c:#include <stddef.h>
edit.c:#include <texttool.h>
edit.c:#include <gno/kerntool.h>
comp.c:#include <stdio.h>
comp.c:#include <stdlib.h>
comp.c:#include <string.h>
comp.c:#include <ctype.h>
comp.c:#include <types.h>
comp.c:#include <shell.h>
comp.c:#include <gsos.h>
comp.c:#include <orca.h>
comp.c:#include <texttool.h>
comp.c:#include <gno/kerntool.h>
link.c:#include <stdio.h>
link.c:#include <stdlib.h>
link.c:#include <string.h>
link.c:#include <ctype.h>
link.c:#include <types.h>
link.c:#include <shell.h>
link.c:#include <orca.h>
link.c:#include <texttool.h>
link.c:#include <gno/kerntool.h>
[39] samples> head link.c

```

{ head displays the first few lines of a file. }

/*

parsing code for compile, cml1, cmlg

The options flags are set up in a bizarre format as follows:

```

76543210 76543210 76543210 76543210
yz      qrstuvwxyz ijklmnop abcdefgh

```

```
[40] samples> set PATH
```

```
path = /dev/orca/utilities:/dev/gno/orca:/dev/gno/utilities
```

```
[41] samples> unset PATH
```

```
[42] samples> set PATH
Variable Not Found
[43] samples> rm m[e][s][s]age[2]
[44] samples> rehash
hashed 0 files.
[45] samples> rm who_cares about_the parameters
{ rm will not be recognized as a command because the hash-table is empty. }
rm: Command not found.
[46] samples> set path = /dev/orca/utilities ~/orca ~/utilities

/dev/orca/utilities /dev/gno/orca /dev/gno/utilities =
[47] samples> set PATH

path = =
[48] samples> set path=~/../orca/utilities ~/orca $HOME/utilities"
{ Filename expansion will be performed before PATH is set. }
[49] samples> set path

path = /dev/orca/utilities /dev/gno/orca /dev/gno/utilities
[49] samples> rehash
hashed 68 files.
```


Appendix B

Prefix Conventions

When **gsh** is started, GS/OS assigns certain values to individual prefixes, and usually the *gshrc* file also sets some prefixes. A total of 32 prefixes are available to the user. The following list documents each prefix and the purpose of each.

<u>Number</u>	<u>Description</u>
@	AppleShare prefix. If GNO resides on an AppleShare volume, this prefix is set to the pathname of the user's directory on the file server; otherwise, this prefix is set to the same pathname as prefix number 9.
*	Boot volume prefix. It is not possible to modify the value of this prefix with the shell's <code>prefix</code> command. The only other way to access this prefix is the GS/OS <code>_GetBootVol</code> call.
0	Prefix 0 is the current working directory. It is the prefix that is changed by the <code>cd</code> command.
1	Directory a program resides in. In the shell, this is usually <code>/bin</code> . The kernel sets this prefix (and 9/) appropriately for each program that is executed.
2-8	Used by Orca 1.0 languages and utilities
2	If the ORCA languages are being used, prefix 2 points to the ORCA Libraries directory.
3	This prefix should point to the directory that contains ORCA.SYS16.
4	ORCA Shell directory should point to the directory that contains the EDITOR, SYSTABS, SYSCMND, etc. files.
5	ORCA Languages directory
6	ORCA Utilities directory
7	ORCA Temp directory - setting this to a RAM disk makes certain editor operations faster.
9	Same as prefix 1/
10-12	Standard input, output, and error device names. GNO sets these to be <code>'.ttyco'</code> .
13	Same as prefix 2/ above if Orca 2.0 languages are being used. .
14	Same as prefix 3/
15	Same as prefix 4/
16	Same as prefix 5/
17	Same as prefix 6/
18	Same as prefix 7/

Appendix C

Errors

gsh tries, when an error occurs, to output an informative error message that will lead you to the solution of your problem. This appendix documents all **gsh** error messages and what the probable cause of the problem might be. There are five classes of errors: generic **gsh**, command-entry, syntax, execution, and built-in. Each error is discussed separately.

Generic **gsh** Errors

These errors can typically occur at any time and may not be directly related to something the user has done. Some of them are trivial, and some are very serious and should be reported immediately.

gsh: There are stopped jobs.

All jobs must be killed before exiting the shell. Use the **jobs** and **kill** commands.

Command Editing Errors

Command editing errors occur when entering information on the command-line. If you try to move the cursor too far to the left or right of your command-line (i.e. before the first character or after the last character), an error will occur. At present, **gsh** indicates a command-entry error with a **Beep**. This is to notify you that the action you requested is not possible.

Syntax Errors

Syntax errors occur while **gsh** is trying to understand the command you have entered on the command-line. Problems arise when you wish to quote an argument (") and only enter one quote.

gsh: Missing ending ".

A second " wasn't supplied when quoting text.

gsh: Missing ending '.

A second ' wasn't supplied when quoting text.

gsh: Too many arguments, so no dessert tonight.

The command-line contained too many arguments which exceeded the available memory allocated by **gsh**.

gsh: Not enough memory for arguments.

No memory was available for allocating command-line arguments.

gsh: Extra '<' encountered.

gsh: Extra '>' or '>>' encountered.

gsh: Extra '>&' or '>>&' encountered

Text may be redirected to only one file.

gsh: No file specified for '<'.

gsh: No file specified for '>' or '>>'.

gsh: No file specified for '>&' or '>>&'.

A file must be specified when redirecting I/O.

gsh: '/' conflicts with '>' or '>>'.

gsh: '/' conflicts with '<'.

Piping is another form of redirection, thus pipes and redirections cannot be mixed.

Execution Errors

After **gsh** parses the command-line, it will then execute the command and pass any arguments to the command. If, however, the command does not exist, **gsh** will report an error. The reason the command does not exist could be either the command name was typed wrong or the command does not exist.

\$0: Command not found.

\$0 represents the command to be executed. Either the command name was entered incorrectly or the command does not exist. Recheck the spelling of the command and check **\$PATH** to make sure the command exists in the pathname list.

\$0: Not executable.

\$0 represents the command to be executed.

heh heh, next time you'll need to specify a command before redirecting.

Redirection was specified but the command-line had no command.

Cannot fork (too many processes?)

An error was encountered forking a process. The most likely culprit is there are too many processes running.

Built-in Command Errors

These are errors which can be returned by many of the built-in commands. Every built-in also contains a Usage message on the proper invocation method.

cd: Not a directory

Tried to change the cwd to a file that isn't a directory.

prefix: could not set prefix, pathname may not exist.

GS/OS Prefix command failed, most likely the pathname did not exist or the disk is damaged.

setdebug: Unknown flag

An unknown flag was sent to **setdebug**. Run **setdebug** with no arguments for a list of possible flags.

ps: error in kvm_open()

ps was unable to access the process data structure. It would be amazing that the kernel is still running for this error to occur.

set: Variable not specified

A variable was not passed to **set**, for example, "**set =bar**". Make sure the variable name was specified without the preceding dollar sign. For example, if **foo** is not set, then "**set \$foo=bar**" would be expanded to "**set =bar**", this resulting in this error.

kill: Invalid signal number

kill: Invalid signal name

See Appendix C for a list of valid signal numbers and names.

fg: No job to foreground.
bg: No job to background.
stop: No job to stop.
There aren't any jobs so this command is useless.

fg: No such job.
bg: No such job.
stop: No such job.
kill: No such job.
The specified job (or process) doesn't exist.

fg: Gee, this job is already in the foreground.
bg: Gee, this job is already in the background.
stop: Gee, this job is already stopped.

Well, this should be self-explanatory. Also, some of these should be impossible to get, unless you're bound and determined to crash gsh, but then, these errors will keep you from crashing it, so, what's the point?

Appendix D

Signals

The following list describes all signals present in the GNO Multitasking Environment. These signals are used to communicate with processes and the GNO kernel. **gsh** provides a means to send signals to processes via the kill command. Furthermore, GNO/ME provides interfaces for C and Assembly Language programmers to handle signals in their programs. Note that not all of these signals are actually used in GNO/ME 2.0.

<u>Name (Value)</u>	<u>Description</u>
SIGHUP (1)	Hangup.
SIGINT (2)	Interrupt.
SIGQUIT (3)	Quit.
SIGILL (4)	Illegal Instruction.
SIGTRAP (5)	Trace trap.
SIGABRT (6)	Abort (cf. abort(3)).
SIGEMT (7)	Emulator trap.
SIGFPE (8)	Arithmetic exception.
SIGKILL (9)	Kill. This signal cannot be caught, blocked, or ignored. If a signal is not specified for the kill command, this signal is sent to the process.
SIGBUS (10)	Bus error.
SIGSEGV (11)	Segmentation violation.
SIGSYS (12)	Bad argument to system call.
SIGPIPE (13)	Write on a pipe or other socket with no one to read it.
SIGALRM (14)	Alarm clock.
SIGTERM (15)	Software termination signal.
SIGURG (16)	Urgent condition present on socket.
SIGSTOP (17)	Stop. This signal cannot be caught, blocked, or ignored.
SIGTSTP (18)	Stop signal generated from keyboard.
SIGCONT (19)	Continue after stop. This signal cannot be blocked.
SIGCHLD (20)	Child status has changed.
SIGTTIN (21)	Background read attempted from control terminal.
SIGTTOU (22)	Background write attempted to control terminal.
SIGIO (23)	Input/Output possible signal.
SIGXCPU (24)	Exceeded CPU time limit.
SIGUSR1 (30)	User defined signal 1.
SIGUSR2 (31)	User defined signal 2.

Appendix E

Non-Compliant Applications

GNO/ME wasn't really designed with the intention of making EVERY program you currently run work under GNO/ME; that task would have been impossible. Our main goal was to provide a UNIX-based multitasking environment; that we have done. We made sure as many existing applications as we had time to track and debug worked with GNO/ME. The current list of compatible and non-compatible applications can be found in the file "RELEASE.NOTES" on the GNO/ME disk.

However, due to the sheer number of applications and authors, there are some programs that just plain don't work; and some that mostly work, except for annoyances such as two cursors appearing, or keyboard characters getting 'lost'. The problem here is that some programs use their own text drivers (since TextTools output was very slow at one time); since GNO/ME doesn't know about these custom drivers, it goes on buffering keyboard characters and displaying the cursor. There is a way, however, to tell GNO/ME about these programs that break GNO/ME's rules.

We've defined an auxType for S16 and EXE files, to allow distinction between programs that are GNO/ME compliant and those that are not. Setting the auxType of an application to \$DC00 disables the interrupt driven keyboard buffering and turns off the GNO/ME cursor. Desktop programs use the GNO/ME keyboard I/O via the Event Manager, and thus should *not* have their auxType changed.

You can change a program's auxType with the following shell command:

```
chtyp -a \ $DC00 filename
```

where filename is the name of the application. As more programmers become aware of GNO/ME and work to make their software compatible with it, this will become less of a problem, but for older applications that are unlikely to ever change (like the America OnLine software) \$DC00 is a reasonable approach.

Appendix F

Termcaps

'Termcap' is short for 'terminal capability', and is the name of a database which applications can use to do full-screen output on any kind of terminal. The termcap database contains records for the various supported terminals, each of which contains fields that look like this:

cap=value

cap is a two-letter code that represents a cursor movement, screen mode change (such as inverse or underline mode), and various other things. **Value** is usually a sequence of control characters that is sent to a terminal to initiate the desired action. **Value** can also be 'boolean', or yes/no, values, for such things as "Does this terminal support cursor movement?". The termcap file is documented in the electronic manual entry **termcap(5)**.

The termcap library does not specifically require GNO/ME.

The following terminal types are supported in the GNO/ME termcap file:

gnocon	GNO Console
CONSOLE	GS/OS .console driver
ptse	Proterm Special Emulation
vt100	DEC VT-100 terminal
ansisys	MS-DOS ANSI.SYS
xerox820	Xerox 820-II CP/M terminal
iw1	Apple ImageWriter I printer
iw-alt	Alternate ImageWriter I printer
deskjet	Hewlett Packard DeskJet 500 printer

The printer entries allow a formatted electronic manual page to be sent to the printer. For example, the following script would bring up the manual page for 'ls', format it for the DeskJet 500, and print it with italics and boldface:

```
set temp=$term
set term=deskjet
man $1 > .ttyb
set term=$temp
```


Glossary

Alias.	A name used as an abbreviation for one or more commands. An alias allows you to replace any command string with a short sequence of characters.
Applesoft.	An implementation of BASIC for the Apple II.
APW.	Apple Programmer's Workshop. Similar to ORCA.
BASIC.	Beginners All-purpose Symbolic Instruction Code. A simple computer language.
Built-in command.	A command processed by gsh . These commands are not external to the shell, but are included within the gsh program.
Command.	An action for gsh to perform. Commands can be either simple or compound. A simple command is an alias assignment, variable assignment, I/O redirection, or built-in command. A compound command is a pipeline.
Directory.	A special type of file that contains a list of other files; usually used to categorize files related in some way.
Environment.	The state of a process, which includes information such as its open files, current directory (working directory), and local and global variables. Three environments exist under gsh : <i>Child environment</i> :: The environment of the child process. <i>Current environment</i> :: The environment of the current process. <i>Parent environment</i> :: The environment of the parent process.
Environment file.	A file that is interpreted by an application to allow the user to customize its operation. For gsh , this file is "gshrc".
Export.	A way to pass a variable from a parent process to child process.
File.	An object used to store data and/or programs. On the IIGS, files are tagged with types such as EXE, SRC, TXT, etc.
Filter.	A command that reads from its standard input and writes to its standard output. For example, a filter program could be written to convert all characters to upper case. Filters are used mainly in pipelines.
Flag.	A character used to represent an option to a command. Flags are either short or long options whose character representations are "-" and "+".
Glob.	Slang for Pathname Expansion.
GNO/ME.	GNO Multitasking Environment. The complete package including the GNO kernel and the GNO Shell.
GNO Kernel.	Heart of GNO/ME. Executes processes when asked by the GNO Shell
GNO Shell.	Provides an interface between the user and the GNO kernel.
gsh.	GNO Implementation of a UNIX-like shell.
GS/OS.	16 bit Operating System for the Apple IIgs.
History.	A variable number of command-lines saved by gsh for future reference. The number of command-lines saved is dependent on the \$HISTORY environment variable.
History file.	A file containing command-lines entered while in a gsh session. The number of command-lines saved is dependent on the \$SAVEHIST environment variable.
Interrupt.	A signal generated by a sequence of keyboard characters or by a command that terminates the current executing process, unless the process has set up a trap to handle the interrupt signal.
I/O Redirection.	The process of changing the standard input, standard output, and standard error associated with a process so that it is redirected to a file instead of the console.
Job.	A set of related processes. A job can be either: <i>Background job</i> :A process that executes with the current process. Background jobs are not associated with the terminal.

Foreground job :	A process that is currently executing and which is associated with the terminal.
Multiprocessing.	Indicates a machine with more than one CPU.
Multitasking.	The ability to run more than one program at a time, or the illusion of more than one program running at a time - usually the latter.
ORCA	Shell programming environment for the Apple //gs. Also a type of whale.
Path search.	The means of searching a pathname list for a command or script.
Pathname.	A string used to identify a file.
	Pathname completion. The means of generating all pathnames matching a given pattern.
	Pathname expansion. The means of replacing a pattern with a list of pathnames matching that pattern.
Pattern.	A string of characters used to match literal characters and/or multiple characters.
Permission.	Each file has certain permissions associated with it: destroy, rename, backup, invisible, write, and read.
Pipe.	A conduit through which a stream of characters can pass from one process to another. This is accomplished by linking the standard output of one process to the standard input of a second process.
Pipeline.	Two or more processes connected together by pipes.
Process.	A single thread of execution that consists of a program and an execution environment.
	Child process: A new process created by another process.
	Parent process: A process that creates a child process.
Process id:	Each active process is uniquely identified by a positive integer called the process id.
ProDOS.	8-bit Disk Operating System for Apple II computers.
Prompt.	A message displayed by gsh when it is ready to receive a command.
Quoting.	A means of including special characters as arguments to a command or as the command name. Certain characters have certain meanings to gsh and quoting them makes gsh ignore them.
Reserved word.	A word that is treated specially by gsh . This word is part of the gsh grammar.
Script.	A sequence of commands contained in a file.
Signal.	An asynchronous message that consists of a number or name that can be sent from one process to another.
Standard Error.	The file associated with error messages for a process. This file is usually the terminal.
Standard Input.	The file associated with a processes input. This file is usually the terminal.
Standard Output.	The file associated with a processes output. This file is usually the terminal.
Tilde Expansion.	Words beginning with "~" are treated specially by gsh . The "~" is expanded to the value of the HOME variable.
UNIX.	Popular operating system which has growing use in education and business. One of the first operating systems to support multitasking.
Variable.	A named location in gsh that contains text. The text of a variable can be expanded in a command by preceding the variable name with a dollar sign (\$).
Wildcard.	See Pattern and Pathname Expansion.
Working directory.	The current directory.

Index

^C 19
\$ 35
^Z 19
\$0 33
\$1 33
\$2 33
\$< 33
\$DC00 50
\$ECHO 34
\$FIGNORE 10, 34
\$HISTORY 9, 34
\$HOME 9, 34
\$IGNOREEOF 34
\$NOBEEP 34
\$NODIREXEC 34
\$NOGLOB 34
\$NONEWLINE 34
\$PATH 31, 34
\$PRECMD 34
\$PROMPT 34
\$PUSHDSILENT 35
\$SAVEHIST 9, 35
\$TERM 35
\$TERMCAP 35
\$USER 35
& 17
. 21
.. 21
.i.OA 26
< 16
> 16
>& 16
>> 16
>>& 16
\n 27, 35
\r 27, 35
\t 27, 35
accelerator 19
alias 15, 30, 34
ansisys 51
AppleShare 45
argument 22
auxType 49
Background 17, 19
backward-char 8, 25
backward-delete-char 8, 25
backward-word 9, 25
BASIC 13
beginning-of-line 9, 25, 26
bg 19, 29
bindkey 7, 25
BLOCKED 29
Bourne shell 3
Built-In Commands 7
built-ins 25
C shell 3
cat 16, 22
cd 26, 45
chdir 26
child process 33
chtyp 50
CLEAR 8, 26
clear-screen 8, 25
cmpl 18, 19
coff 14, 17
command arguments 7
command name 7, 15
command-line 3, 7, 11, 15, 17, 19
command-line buffer 15
command-line editor 7, 9, 10, 25, 26
commands 26
complete-word 10, 25
CONSOLE 51
cp 18, 20, 22, 33
CTRL-A 9, 26
CTRL-B 8
CTRL-D 8, 9, 10
CTRL-E 9
CTRL-F 8
CTRL-K 26
CTRL-L 8
CTRL-N 10
CTRL-P 10
CTRL-R 8
CTRL-X 8
DELETE 8
delete-char 9, 25
deskjet 51
Desktop 50
df 26
dirs 28
done 19
DOWN-ARROW 10
down-history 10, 25
echo 7, 16, 24, 27
~ 21
editor 4, 5
end-of-line 9, 25
Errors 46
ESC 11, 26
ESC-E 11
ESC-Y 26
Event Manager 50

EXE 5, 7, 25, 33, 34, 50
EXEC 5, 7
exit 27
export 33
fg 19, 29
filter processes 17
foreground 19
forward-char 8, 25
forward-word 9, 25
globbing 22, 34
GNO Installer 3, 4, 9
GNO Kernel 3, 4, 5, 17, 25, 49
GNO shell 3, 4, 5, 9, 15, 17
GNO/ME 3, 4, 5, 13, 16, 17, 19, 49, 50
gnocon 35, 50
grep 17, 22
GS/OS 16
gsh 3, 5, 7, 9, 10, 11, 15, 16, 18, 19, 21, 22, 24, 26, 33, 49
gshrc 3, 4, 5, 9, 11, 15, 33
GUI 3
hash 31
HFS 23, 34
HISTORY 9, 27, 34
history buffer 9
home directory 21
I/O 50
insert 9, 11
iw-alt 51
iw1 51
job 19
Job Control 19
job number 17
jobs 29
kern 5
kernel 18, 26
kill 19, 29
kill-end-of-line 9, 25
kill-whole-line 8, 25
Korn shell 3
LEFT-ARROW 8
less 17
list-choices 10, 25
login(1) 35
long options 13, 14
ls 13, 14, 16, 18, 19, 22, 33, 51
man 13, 14, 17
manual pages 13
mkdir 22
modem 16
more 13, 17, 19, 23
MouseText 33
multitasking 17, 19, 33
mv 33
NEW 29
newline 8, 25
NullProcess 18
OA 7, 11, 26
OA-< 9, 26
OA-> 9
OA-CLEAR 26
OA-D 9
OA-E 9, 11
OA-LEFT ARROW 9
OA-RIGHT-ARROW 9
OA-Y 26
OMF 17
Open Apple 7
option arguments 13
ORCA 4, 5, 15, 16
ORCA shell 3, 17
overstrike 9
paging 17
parameters 7
parent process 33
PATH 4, 5, 24
path delimiter 4
Pathname Expansion 22
pathname separator 4
PAUSED 30
performance 19
pipeline 16
popd 28, 35
pre-emptive multitasking 17
prefix 31
Prefix Conventions 45
printer 16
process 19
process ID 17
ProDOS 13, 23
prompt 5
ps 18, 19, 20, 29
ptse 51
pushd 28, 35
pwd 28
qtime 7
quote 35
raw-char 25, 26
READY 29
redirect 16
redirection 16
redirection operators 16
redisplay 25
redraw 8
regular expressions 22
rehash 24, 31

RELEASE.NOTES 50
Remote Access 11
RETURN 8
RIGHT-ARROW 8
running 19, 29
S16 50
Sample gsh session 39
SAVEHIST 9
Scope 33
script 11, 24, 28
serial port 16
set 31, 35
setdebug 30
setenv 31
shell script 33
short options 13, 14
SIGABRT 49
SIGALRM 49
SIGBUS 49
SIGCHLD 49
SIGCONT 49
SIGEMT 49
SIGFPE 49
SIGHUP 49
SIGILL 49
SIGINT 49
SIGIO 49
SIGKILL 49
SIGNAL 25, 29
signals 49
SIGPIPE 49
SIGQUIT 49
SIGSEGV 49
SIGSTOP 49
SIGSYS 49
SIGTERM 49
SIGTRAP 49
SIGTSTP 49
SIGTTIN 49
SIGTTOU 49
SIGURG 49
SIGUSR1 49
SIGUSR2 49
SIGXCPU 49
source 5, 28
standard error 16, 18
standard input 16, 33
standard output 16, 18
stop 19, 30
stopped 19
storage device 16
SUSPENDED 30
SYS16 5, 7, 34

TAB 10
TERM 33
termcap 35, 50
terminal 11, 16, 50
terminal emulation 35
TextTools 50
tilde 21
toggle-cursor 9, 25
tset 28, 35
ttya 16
unalias 15, 31
undefined-char 25, 26
unhash 31
UNIX 5, 14, 34, 50
unset 35
UP-ARROW 10
up-history 10, 25
variables 3, 33, 35
vi 4
vt100 51
WAITING 30
WAITSIGCH 30
which 28
wildcards 22
word 7
xerox820 51

