AT&T

Issue 1
October 1984

AT&T 3B2 Computer UNIX™ System V Release 2.0 User Environment Utilities Guide

Select Code
305-426

Comcode
403778400

TRADEMARKS

The following trademark is used in this manual.

- UNIX — Trademark of AT&T Bell Laboratories.

NOTICE

NOTE

This Utilities Guide contains descriptive information and UNIX* System manual pages for the commands included in one of the utilities provided with your 3B2 Computer. Since this utilities is provided with the 3B2 Computer, the manual pages have already been filed in the *3B2 Computer UNIX System V User Reference Manual.* If you do not need duplicate copies of these manual pages, they may be discarded.

A UTILITIES binder is provided with the 3B2 Computer for you to keep the descriptive information from all the Utilities Guides together. Remove the descriptive information from the soft cover, place the provided tab separator in front of the title page, and file this material in the UTILITIES binder. As previously mentioned, UNIX System manual pages may be destroyed.

If you ordered extra copies of this Utilities Guide, they should be left in the individual soft covers.

---

* Trademark of AT&T Bell Laboratories

AT&T

Issue 1
October 1984

AT&T 3B2 Computer UNIX™ System V Release 2.0 User Environment Utilities Software Information Bulletin

Select Code
305-363

Comcode
403778228

NOTE

This Software Information Bulletin (SIB) should be filed in the *3B2 Computer Owner/Operator Manual.* A tab separator, labeled " SOFTWARE INFORMATION BULLETINS," has been placed at the back of the *Owner/Operator Manual* in order to provide a convenient place for filing SIB's. Place the tab separator provided with this SIB in front of the title page and file this material behind the SOFTWARE INFORMATION BULLETINS tab separator in the *Owner/Operator Manual.*

USER ENVIRONMENT SOFTWARE INFORMATION BULLETIN

INTRODUCTION

This Software Information Bulletin provides important information concerning the User Environment Utilities. Please read this bulletin carefully before attempting to install or use these utilities.

The AT&T 3B2 Computer User Environment Utilities are for use by all users. They are part of the UNIX* System V Release 2.0 configuration provided with all 3B2 Computers.

IMPROVEMENTS SINCE PRIOR RELEASE

The User Environment Utilities now include the commands which were in the previous release, plus commands that were in *Calculator Utilities* and *Shell Programming Utilities.* A few other commands have also been added.

SOFTWARE DEPENDENCIES

The User Environment Utilities require that the Directory and File Management Utilities already be loaded on the 3B2 Computer.

---

* Trademark of AT&T Bell Laboratories

NOTES ON USING UTILITIES

Command Access

To use these commands, you must be logged in on the system, have the Directory and File Management Utilities loaded, and have the User Environment Utilities loaded.

DOCUMENTATION

This Software Information Bulletin should be inserted in the *3B2 Computer Owner/Operator Manual.*

The user environment commands provided by the User Environment Utilities are described in the *3B2 Computer User Environment Utilities Guide.*

RELEASE FORMAT

Storage Structure

The User Environment Utilities (commands) are installed in the **/bin** and **/usr/bin** directories.

System Requirements

The minimum equipment configuration required for the use of the User Environment Utilities is 0.5 megabytes of random access memory and a 10-megabyte hard disk.

To install the User Environment Utilities software, there must be 172 free blocks in the **root** file system and 668 blocks in the **usr** file system. Adequate storage space is checked automatically as part of the installation process. The installation process installs the utilities only if adequate storage space is available.

The User Environment Utilities for the 3B2 Computer are distributed on one floppy disk. Refer to the *3B2 Computer User Environment Utilities Guide* for further information on the commands.

Files Delivered

The User Environment Utilities are delivered on a single floppy disk. The directory structure and files are as follows.

DIRECTORY	FILES		
/boot	sxt.o		
/etc/master.d	sxt		
/bin	basename dirname env	line nice nohup	time tty
/usr/bin	at banner batch bc cal	calendar crontab dc factor logname	shl tabs units xargs
/usr/lib	calprog	lib.b	unittab
/usr/options	calc.name	shell.name	usrenv.name

UTILITIES INSTALL PROCEDURE

Use the standard software installation procedure described in the *3B2 Computer Owner/Operator Manual* for the installation of the User Environment Utilities.

Note that this and several other utilities involve the installation of one or more drivers (included on the utilities floppy disk). The other driver associated utilities include, but are not limited to:

- AT&T 3BNET
- Inter-Process Communication
- Performance Measurements.

The computer displayed install instructions for these driver associated utilities direct you to quit the install procedure and execute a "shutdown" command.

The specified shutdown options bring the 3B2 Computer to the firmware mode and automatically reboot the system. During this sequence, a new **/unix** is generated which includes the new driver(s).

When installing more than one of these driver associated utilities at the same time, you can (but it is not necessary to) execute the quit and shutdown sequence after each utilities is installed. The quit and shutdown sequence only needs to be executed after the last utilities is installed.

UTILITIES REMOVE PROCEDURE

Use the standard software remove procedure described in the *3B2 Computer Owner/Operator Manual* for the removal of the User Environment Utilities.

CONTENTS

Chapter 1

INTRODUCTION

PAGE

Chapter 1

INTRODUCTION

GENERAL

This guide describes command syntax and use of the User Environment Utilities available with your AT&T 3B2 Computer.

The 3B2 Computer user operates in a predefined executing environment. This environment is defined when a user logs in. The login process establishes environment variables, home directory, path, etc. A user may optionally add or alter environment variables in a **.profile** in his home directory.

Many User Environment Utilities commands allow users to control their inherited environment. This allows the user to schedule commands to be executed at a specific time, or access more than one shell from a single terminal.

Some of the commands are useful in shell programming. The "machid" commands allow you to tailor shell programs to run in

several UNIX* System machine environments. For instance, a shell program may be written that would allow the user to perform a specific function based on the type of computer. This capability increases the portability of shell programs.

Four of the commands can be used to perform arithmetic calculations. These commands can be executed directly, or used inside a file.

GUIDE ORGANIZATION

This guide is structured so you can easily find desired information without having to read the entire text. The remainder of this document is organized as follows:

- Chapter 2, "COMMAND DESCRIPTION," describes the command formats (syntax) for each command in the User Environment Utilities. The descriptions include the purpose of the command, a discussion of the command syntax and options, and examples of using each command.

- Appendix, "MANUAL PAGES," contains the User Environment Utilities UNIX System manual pages.

* Trademark of AT&T Bell Laboratories

Chapter 2

COMMAND DESCRIPTIONS

PAGE

Chapter 2

COMMAND DESCRIPTIONS

GENERAL

The User Environment Utilities provide 27 UNIX System commands. A summary of these commands is provided in Figure 2-1.

The User Environment commands are useful when:

- Performing mathematical calculations
- Writing shell programs
- Checking or changing executing environment of commands
- Scheduling commands to be executed at a later time.

COMMAND	DESCRIPTION
at	Used to execute commands at a specified time.
banner	Creates a banner with large letters.
basename	Prints only the last name of a path name argument.
batch	Used to execute commands when the system load level permits.
bc	Used to perform precise arithmetic calculations.
cal	Prints a calendar for a specified month and/or year.
calendar	Invokes a user's reminder service.
crontab	Used to schedule command execution using **cron**.
dc	Used to perform precise arithmetic calculations using a stack to keep a record of the previous calculation.
dirname	Prints all but the last name of path name arguments.
env	Executes a command with modified environment variables.
factor	Used to calculate the prime factors of any number.
line	Reads a line from the standard input and copies it to the standard output.

Figure 2-1. User Environment Commands (Sheet 1 of 3)

COMMAND	DESCRIPTION
logname	Displays the environment variable **$LOGNAME** assigned to the user upon login.
nice	Runs a command at a lower priority level.
nohup	Makes a command immune to hangups and quits.
pdp11*	Used to determine if you are on a PDP-11/45 or PDP-11/70 computer.
shl	Allows a user to interact with several shells from the same terminal.
tabs	Sets tab stops on a printer or terminal.
time	Times a command to see how long it takes for the command to execute.
tty	Prints the path name of the user's terminal.
u370*	Used to determine if you are on an IBM 370 computer.
u3b*	Used to determine if you are on a 3B20S computer.
u3b5*	Used to determine if you are on a 3B5 computer.

Figure 2-1. User Environment Commands (Sheet 2 of 3)

COMMAND	DESCRIPTION
units	Used to find the conversion factor between two different standard unit values.
vax*	Used to determine if you are on a VAX-11/750 or VAX-11/780 computer.
xargs	Used to execute a command or a shell program one or more times by combining arguments to this command with arguments read from the standard input.

Note: The **pdp11, u370, u3b, u3b5,** and **vax** commands are referred to as the **machid** commands when discussed later in this chapter.

Figure 2-1. User Environment Commands (Sheet 3 of 3)

HOW COMMANDS ARE DESCRIBED

A common format is used to describe each of the commands. The format is as follows:

- **General:** The purpose of the command is defined. Any unique or special information about the command is also provided.
- **Command Format:** The basic command format (syntax) is defined and the various arguments and options discussed.
- **Sample Command Use:** Example command line entries and system responses are provided to show you how to use the command.

In the command format discussions, the following symbology and conventions are used to define the command syntax.

- The basic command is shown in bold type. For example: **command** is in bold type.
- Arguments that you must supply to the command are shown in a special type. For example: **command** *argument*
- Command options and arguments that do not have to be supplied are enclosed in brackets ([]). For example: **command** *[optional arguments]*
- The pipe symbol (¦) is used to separate arguments when one of several forms of an argument can be used for a given argument field. The pipe symbol can be thought of as an exclusive OR function in this context. For example: **command** *[argument1 ¦ argument2]*

In the sample command discussions, the lines that you input are ended with a carriage return. This is shown by using *<CR>* at the end of the lines.

Refer to the Appendix or the *3B2 User Reference Manual* for UNIX System V manual pages supporting the commands described in this guide.

The following conventions are used to show your terminal input and the system output.

This style of type is used to show system generated responses displayed on your screen.

This style of bold type is used to show inputs entered from your keyboard that are displayed on your screen.

These bracket symbols, < > identify inputs from the keyboard that are not displayed on your screen, such as: <CR> carriage return, <CTRL d> control d, <ESC g> escape g, passwords, and tabs.

This style of italic type is used for notes that provide you with additional information.

COMMAND SUMMARY

at — Execute Command at a Specified Time

General

The **at** command is used to execute one or more commands at a specified time and date. Output from the at command is mailed to the user unless it is redirected to a file, printer, etc.

Command Format

The **at** command has the following format:

at *time [date] [+ increment]*
at *-r job...*
at *-l [job...]*

The *time* argument identifies the time of day you want the at-file run. A 24-hour clock is assumed unless the optional letters, **A** (AM), **P** (PM), **N** (noon), or **M** (midnight) follow the four digit number.

The *date* argument can be a month followed by a space and then the day, or just the name of the day of the week. You can also specify **week** which means schedule for seven days from today's date.

The + *increment* argument allows you to schedule commands to be executed according to a time interval. The increment is a number followed by one of the following intervals: minutes, hours, days, weeks, months, or years.

The *-l* option lists all the jobs you have previously scheduled with the **at** or **batch** command. **at** job numbers are ended with a **.a** and **batch** job numbers with a **.b**.

The *-r* option removes jobs previously scheduled by the **at** or **batch** commands.

Sample Commands

The following examples show how to enter an **at** command to execute the file " filename" at a specified time.

```
$ at 11:15 Aug 6<CR>
nroff -mm memo > memo.f<CR>
<CTRL d>
job 460653300.a at Mon Aug 6 11:15:00 1984
$
```

```
$ at 1350 < filename > outputfile<CR>
job 460653300.a at Mon Aug 6 11:15:00 1984
$
```

```
$ at -r 460653300.a<CR>
$
```

banner — Make Banners

General

The **banner** command prints a banner of the arguments specified in the command line. Each of the arguments can be up to 10 characters long. Quotation marks can be used to force the output onto one line.

Command Format

The **banner** command has the following format:

banner *arguments*

arguments cannot be more than 10 characters. There is no limit to the number of arguments that can be specified.

Sample Command

The following example shows how to enter the **banner** command and the response that would follow:

```
$ banner "1 2 3" <CR>

  #          #####          #####
 ##         #     #        #     #
# #                #              #
  #          #####          #####
  #         #                     #
  #         #               #     #
#####       #######          #####

$
```

basename — Prints Portions of Path Names

General

The **basename** command is mainly used in shell scripts to output the last name in a path name. It is usually enclosed in grave accents (‘command *arguments*‘) to indicate command substitution. (Note that grave accents are also called back quotes.)

Command Format

The **basename** command has the following format:

basename *string* [*suffix*]

The *string* argument identifies the path name that is to be processed. The command deletes any prefix ending in a slash (/). The pattern identified by the *suffix* argument is also deleted from the *string* argument if the pattern is present at the end of the *string*.

Sample Commands

The following command line entries and system responses show the basic operation of **basename** command. The **pwd** command prints the current directory path. The output of the pwd is then used as an argument to basename and only the last name in the path is printed. The **basename** command is then used with a *suffix* argument to delete the letter **s** from the end of the resulting name.

```
$ pwd<CR>
/f1/fred/letters
$ basename `pwd`<CR>
letters
$ basename /f2/joan/letters s<CR>
letter
$
```

Another example of using the **basename** command is using it in a shell script (c). In this example, *x* is a variable which is set to the basename of the given path name and then echoed.

```
$ x=`basename /f3/memos/newsletter`<CR>
$ echo $x<CR>
newsletter
$
```

batch — Execute Commands at a Later Time

General

The **batch** command is used to execute commands at a later time when the system load level permits. It is very useful for running text processing commands such as *nroff* or *troff,* or for compiling C programs, which require a large amount of processor time. The output from the **batch** command is mailed to the user unless it is redirected to a file.

Command Format

The **batch** command accepts input directly or given in a file. The format of the **batch** command, when data is entered directly, is as follows:

batch
command lines
<CTRL **d**>

The *command lines* argument represents the commands you wish to execute.

The format of the **batch** command, when data is given in a file is as follows:

batch < *file*

The *file* argument identifies the file containing the commands you want to execute.

To list the jobs you have previously scheduled, use the **at -l** command. To remove jobs, use the **at -r** command.

Sample Command

The following command line entry shows how to enter an *nroff* command, using the **batch** command and the response that would follow.

```
$ batch<CR>
nroff file1 > file2<CR>
<CTRL d>
job 460657390.b at Mon Aug 6 12:35:01 1984
$
```

bc — Calculator

General

The **bc** command is used to do basic arithmetic, number base conversions, trigonometric functions, exponential calculations, natural logarithm calculations, and Bessel functions. It reads input from file arguments given and then reads standard input.

The **bc** command is actually a preprocessor for the **dc** command. The output from the **bc** command is piped to **dc** unless the **-c** (compile only) option is present.

The **bc** command has special operators and functions which must be used when performing any calculations. These operators and functions are described in the following list.

+	adds two numbers and displays the result.
-	subtracts two numbers and displays the result.
/	divides the left argument by the right argument and displays the result.
*	multiplies two numbers and displays the result.
%	displays the remainder from a division process.
^	raises the left argument to a power denoted by the right argument.
a	displays the arctangent of the right argument.
c	displays the cosine of the right argument.
s	displays the sine of the right argument.
e	displays the exponential of the right argument.

l displays the natural logarithm of the right argument.

ibase changes the input base.

obase changes the output base.

scale changes the number of digits to the right of the decimal point.

quit exits the program.

Command Format

Input for the **bc** command can be entered directly or given in a file. The format of the **bc** command, when data is entered directly, is as follows:

bc
calculation

The *calculation* argument represents the mathematical calculation you wish to execute.

The format of the **bc** command, when data is given in a file, is as follows:

bc [*-c*] [*-l*] *file*

The *-c* (compile only) option causes the preprocessed output from **bc** to be written to standard output instead of piping it to **dc**. No calculations are performed.

The *-l* option passes the input through an arbitrary precision math library. This library contains the sine, cosine, exponential, log, arctangent, and Bessel functions. See examples in "Trigonometric and Exponential Calculations."

The *file* argument identifies the file you want the **bc** command to act upon. Information on variables, operators, and statements you can use in these files is given in the manual pages.

Sample Calculations

The following examples show how to use the **bc** command for most commonly used calculations. To do these examples, first be sure you have the system prompt (**$**).

Addition, Subtraction, Multiplication, and Division

In this example, you want to add 5 plus 5.

```
$ bc<CR>
5+5<CR>
10
quit<CR>
$
```

This format is also used for subtraction, multiplication, and division. The only difference is you have to use the appropriate operator.

Calculations with Negative Numbers

To do calculations with negative numbers, you must precede the negative number with a minus sign (**-**). In this example, you want to add a negative 6 to a positive 3 (a positive number will not have a + sign preceding it).

```
$ bc<CR>
-6+3<CR>
-3
quit<CR>
$
```

Find the Remainder of a Division Calculation

The **bc** command rounds off the remainder when performing division calculations. To find the remainder, you need to use the remainder **%** operator. In this example, you want to find the remainder of the problem (17 divided by 5). Notice that only the remainder is displayed.

```
$ bc<CR>
17%5<CR>
2
quit<CR>
$
```

Raise a Number to a Power

In this example, you want to raise 2 to the 6 power.

```
$ bc<CR>
2^6<CR>
64
quit<CR>
$
```

Find the Square Root

In this example, you want to find the square root of 49.

```
$ bc<CR>
sqrt(49)<CR>
7
quit<CR>
$
```

Combining Calculations

When combining several types of calculations into one problem, the **bc** command performs the calculations according to the following rules.

1. All calculations enclosed in parenthesis are performed first. If any calculations enclosed in parenthesis are located within another set of parenthesis, the innermost calculations are always performed first.

2. All calculations involving square root and raising a number to a power are performed next. These calculations are performed from right to left.

3. After performing all calculations enclosed in parenthesis, multiplication and division calculations are performed. These calculations are performed from left to right.

4. Unenclosed addition and subtraction calculations are always performed last. These calculations are performed from left to right.

With **bc**, you can do many calculations with one input. In this example, you want to evaluate 5 times 4 plus 3 times (2 + 2) times 2 to the third power and divide the entire calculation by 2.

```
$ bc<CR>
(5*4+3*(2+2)*2^3)/2<CR>
58
quit<CR>
$
```

Continuous Calculations

You can keep doing calculations with **bc** after each answer without having to quit **bc**. You can also change the type of calculation being performed without quitting **bc**. In this example, you have to do five problems: add 30 to 10, add 6 to 3, multiply 2 times 6, multiply 3 times 8, and add 3 to the sum of 2 times 6. The following example shows how to perform all five calculations before quitting **bc**:

```
$ bc<CR>
30+10<CR>
40
6+3<CR>
9
2*6<CR>
12
3*8<CR>
24
3+2*6<CR>
15
quit<CR>
$
```

Changing the Accuracy of Calculations

You can change the accuracy of calculations by increasing the number of digits after the decimal point. In **bc**, this is known as changing the **scale**. The scale in **bc** is initially set to 0. After changing the scale, to get **bc** back to a scale of 0 you must change the scale to 0. In this example, you want to divide 100 by 3 with an accuracy of 5 digits after the decimal point.

```
$ bc<CR>
scale=5<CR>
100/3<CR>
33.33333
scale=0<CR>
quit<CR>
$
```

Changing the Input Base

The **bc** command is normally set for a base of 10 (decimal). You can change the input base from base 10 to base 8 (octal), or base 16 (hexadecimal). To return **bc** to base 10, you must change the input base to 10. You convert the input base to find the base 10 equivalent of a number in base 8, or base 16.

In this example, you want to find the base 10 equivalent of 1000 in base 16 and then convert back to base 10 using the command "ibase=A".

```
$ bc<CR>
ibase=16<CR>
1000<CR>
4096
ibase=A<CR>
1000<CR>
1000
quit<CR>
$
```

Changing the Output Base

The **bc** command is normally set for base 10 (decimal). You can change the output base from base 10 to base 8 (octal), or base 16 (hexadecimal). To return **bc** to base 10, you must change the output base to 10. You convert the output base to find the base 8, or base 16 equivalent of a base 10 number.

In this example, you want to find the base 16 equivalent of 1000 in base 10 and then change back to base 10 using the command "obase=A".

```
$ bc<CR>
obase=16<CR>
1000<CR>
3E8
obase=A<CR>
1000<CR>
1000
quit<CR>
$
```

Using Registers in Calculations

You can perform calculations using one or more of 26 registers designated **a** through **z**. Registers can be used in any of the following types of calculations:

- Addition
- Subtraction
- Multiplication
- Division (positive and whole number results are required when you use more than one register)
- Raise a number (register) to a power
- Multiple calculations using a parenthetical expression.

You can do more than one calculation at a time using register(s). Once you have done a calculation using a register, the result of that calculation will be the content of that register.

You must be in the **bc** command to use these registers. You enter information into a register by typing the register designator (**a** through **z**), an equal sign (=), the information (or calculation), and a carriage return. To see the information or result of a calculation, you enter the register designator followed by a carriage return. The system will display the content of the register.

You can also do calculations with a register using the **bc -l** command. However, you can only use one register.

In the following sample calculations you want to enter 4 into the **x** register and 2 into the **y** register. Then check the contents of the register before performing the sample calculations.

Inputting 4 in the **x** register and 2 in the **y** register:

```
$ bc<CR>
x=4<CR>
y=2<CR>
x<CR>
4
y<CR>
2
z=x+y<CR>
z<CR>
6
z=y-x<CR>
z<CR>
-2
z=x*y<CR>
z<CR>
8
z=x/y<CR>
z<CR>
2
z=x^y<CR>
z<CR>
16
z=(x+x)*y<CR>
z<CR>
32
quit<CR>
$
```

Trigonometric and Exponential Calculations

The **bc** command has an option (**-l**) which accesses the arbitrary precision math library. This option can be used with the **bc** command to:

- Find the sine of an angle
- Find the cosine of an angle
- Find the arctangent of an angle
- Find the natural logarithm (ln) of a number
- Raise a number to the **e** (2.718) power

The **bc -l** command performs all calculations involving angles in radians. Therefore, if the angle you want to work with is in degrees, you must change it to radians (360 degrees = 2 pi radians). To change degrees to radians, multiply the number of degrees by 0.01745.

Find the Sine of 45 Degrees.

```
$ bc -l<CR>
45*.01745<CR>
0.78525
s(.78525)<CR>
.707002
quit<CR>
$
```

Find the Cosine of 1.04700 Radians (60 Degrees).

```
$ bc -l<CR>
c(1.04700)<CR>
.500171
quit<CR>
$
```

Find the Natural Logarithm of the Number 20.

```
$ bc -l<CR>
l(20)<CR>
2.995732
quit<CR>
$
```

Find the Exponential of the Number 2.

```
$ bc -l<CR>
e(2)<CR>
7.389056
quit<CR>
$
```

Find the Sine of 45 Degrees using a register.

```
$ bc -l<CR>
x=45*0.01745<CR>
s(x)<CR>
.707002
quit<CR>
$
```

cal — Print Calendar

General

The **cal** command prints a calendar for the year specified. If an argument for a month is also specified, a calendar for only that month will be printed.

Command Format

The **cal** command has the following format:

cal *[[month] year]*

month must be a number ranging from 1 through 12

year must be a number ranging from 1 through 9999.

If no arguments are given, a calendar for the current month will be printed.

Sample Command

The following example shows how to enter the **cal** command and the response that would follow:

```
$ cal 3 1984<CR>
   March 1984
 S  M Tu  W Th  F  S
              1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

$
```

calendar — Reminder Service

General

The **calendar** command is used to access a file in the current directory named *calendar* and output the lines that contain today's or tomorrow's date. The **calendar** command can be automatically executed on a daily basis as a function of the **cron** command. When automatically executed, the **calendar** command accesses every *calendar* file on the system and reports any results to the appropriate user via **mail**. To receive reminders automatically via **mail**, the *calendar* file should be created in your login directory.

The lines in the *calendar* file must contain dates in some reasonable form. The following is an example of a *calendar* file located in your login directory.

```
$ cat calendar<CR>
Confirm travel reservations on 05/03/84.
Bill's 10th anniversary on June 27, 1984.
Appointment with Dr. Miller on 6/28/84 at 9:00 a.m.
Nancy's birthday on Oct 29.
$
```

Each of the lines containing a date will be output via mail on the day preceding and the day indicated.

Command Format

The **calendar** command has the following format:

calendar [-]

The **-** argument is an optional flag that causes the command to execute for every user having a *calendar* file. This is the form of the command that is executed by the **cron** command to check all

calendar files on a daily basis. Without the argument, the **calendar** command will only check your **calendar** file and report its contents.

Sample Command

The following example shows how to enter the **calendar** command and the output that would follow if the current date was June 27, 1984:

```
$ calendar<CR>
Bill's 10th anniversary on June 27, 1984.
Appointment with Dr. Miller on 6/28/84 at 9:00 a.m.
$
```

In order for this example to be correct, the *calendar* file must be located in the current working directory.

In order to provide reminder service automatically, an entry must be made in the crontab file for **calendar**. The following is a sample crontab entry for **calendar**:

0 9 * * * /usr/lib/calendar -

With this entry, the **calendar** command would execute every morning at 9:00 a.m. (provided the system is up) and search all login directories for *calendar* files. For a detailed description of the crontab file and how its entries are structured, refer to the *3B2 Computer System Administration Utilities Guide.*

crontab — Clock Used to Schedule Commands

General

The **crontab** command allows specified users to submit their jobs for execution. Users are permitted to use crontab if their name is in the file **/usr/lib/cron/cron.allow**. Users are denied permission if their name is in the file **/usr/lib/cron/cron.deny**. If neither file exists, only root is allowed to use the crontab command.

The **crontab** command accepts a command file or standard input. The input file normally contains lines that are broken into six fields. Each field is separated by a space. The first five fields are dedicated to:

minute (0-59)
hour (0-23)
day of the month (1-31)
month of the year (1-12)
day of the week (0-6 with Sunday=0)

The last field is the command you want to execute. If no file is specified, the data is read from the standard input.

Commands in a crontab file are executed in a user's home directory. The output from the command is mailed to the user unless redirected. Therefore, you would normally want to redirect the output to a file.

The crontab command is especially useful for scheduling programs to run periodically, such as accounting programs, weekly reports, or system usage programs.

Command Format

The **crontab** command uses the following formats:

crontab *[-l] [-r] [file]*

[file] contains the command lines a user wishes to have executed by cron. If the file is not specified, command lines are read from standard input.

The *-l* option lists the crontab files for the invoking user.

The *-r* option removes the users files from the crontab directory.

None of the options for the **crontab** command can be used together.

Sample Commands

The following is an example of the **crontab** command using a command file as the input.

Create the command file (whoison) containing the following line:

0 8 * * 1-5 who ;ps >> outputfile

This line instructs cron to execute a " who" command and a " ps" command every Monday through Friday at 8 AM and place the output in a file named " outputfile" in your home directory. To have cron execute the command file you would enter the command:

```
$ crontab whoison<CR>
$
```

To have cron execute the same command using standard input, you would enter the following commands:

```
$ crontab<CR>
0 8 * * 1-5 who ;ps >> outputfile<CR>
<CTRL d>
$
```

To list all the commands in your crontab file, you would use the **l** option. For example:

```
$ crontab -l<CR>
0 8 * * 1-5 who ;ps >> outputfile
$
```

dc — Calculator

General

The **dc** command does various arithmetic calculations on one or two numbers that have been placed on a pushdown stack. The stack is like that of a calculator using reverse Polish notation. The **dc** command takes one or two numbers from the top of the stack, performs the desired arithmetic operation, and returns the number(s) to the stack.

dc Operators and Functions

+	adds the top two numbers on the stack and stores the result in their place.
-	subtracts the top two numbers on the stack and stores the result in their place.
*	multiplies the top two numbers on the stack and stores the result in their place.
/	divides the top two numbers on the stack and stores the result in their place.
%	stores the remainder of a division operation in the top of the stack.
^	raises the second number from the top of the stack to the number in the top of the stack and stores the result in their place.

> ***Note:*** When using +, **-**, *****, /, **%**, or ^, an exponent must not have any digits after the decimal point.

v restores the top number of the stack with the square root of that number.

s*x* stores the number on the top of the stack in a register named *x* (where *x* may be any character). If **s** is uppercase, *x* is treated as a stack.

l*x* stores the number in register *x* on the stack. If the **l** is uppercase, register *x* is treated as a stack and its top value is placed on the main stack.

> ***Note:*** All registers start with an empty value which is treated as a zero by the command **l** and as an error by the command **L**.

d duplicates the top value on the stack.

p prints the top value on the stack.

f prints all values on the stack and in the registers.

x removes the top element from the stack. Treats it as a character string and executes it as a string of dc commands.

[...] puts the bracketed character string on the top of the stack.

q exits the program.

Command Format

The **dc** command accepts data entered directly or from file arguments. The format of the **dc** command when data is entered directly is as follows:

dc
data on stack
operation symbol
p

The *data* argument represents any numbers you have input or numbers existing in the stack.

The *operation symbol* argument represents the mathematical calculation you wish to execute.

The **p** prints the character on top of the stack.

The format of the **dc** command when data is given from a file is as follows:

dc *file*

The *file* argument identifies the file that you want input to the **dc** command.

Sample Calculations

The following examples show how to use the **dc** command when the data is entered directly.

Addition, Subtraction, Multiplication, and Division

In this example, you want to add 10 and 20.

```
$ dc<CR>
10<CR>
20<CR>
+<CR>
p<CR>
30
q<CR>
$
```

This format is also used for subtraction, multiplication, and division. The only difference is you have to use the appropriate operation symbol.

Calculations with Negative Numbers

To do calculations with negative numbers, you must precede the negative number with a minus (-) sign. In this example, you are adding a negative 6 to a positive 3 (a positive number does not need a + preceding it.)

```
$ dc<CR>
-6<CR>
3<CR>
+<CR>
p<CR>
-3
q<CR>
$
```

Find the Remainder of a Division Calculation

When performing division calculations, the **dc** command rounds off the remainder. To find the remainder of a division calculation, you need to use the remainder (**%**) calculation symbol instead of the division symbol. In this example, you want to find the remainder of the problem (25 divided by 3).

```
$ dc<CR>
25<CR>
3<CR>
%<CR>
p<CR>
1
q<CR>
$
```

Raise a Number to a Power

In this example, you are raising the number 2 to the 6th power.

```
$ dc<CR>
2<CR>
6<CR>
^<CR>
p<CR>
64
q<CR>
$
```

Find the Square Root of a Number

In this example, you want to find the square root of 49.

```
$ dc<CR>
49<CR>
v<CR>
p<CR>
7
q<CR>
$
```

Note: Square root answers are rounded off to the nearest whole number.

Combining Calculations

In this example, you are executing a string of different types of calculations without quitting between each one. Also, notice that the answer is not displayed until it is requested by entering **p**.

```
$ dc<CR>
12<CR>
6<CR>
*<CR>
3<CR>
-<CR>
25<CR>
5<CR>
/<CR>
+<CR>
p<CR>
74
q<CR>
$
```

Continuous Calculations

The **dc** command allows you to perform separate calculations continuously without having to return to the UNIX System. You can also change the type of calculation being performed. In this example, you are executing two completely separate calculations without having to quit **dc**.

```
$ dc<CR>
30<CR>
10<CR>
+<CR>
p<CR>
40
c<CR>
5<CR>
6<CR>
*<CR>
p<CR>
30
q<CR>
$
```

Converting Number Base

Normally, **dc** is set for base 10 (decimal). You can set the input and output bases in **dc** to base 10 (decimal), base 8 (octal), or base 16 (hexadecimal). By changing the input base, you can do calculations in the different bases. By changing the output base you can do calculations in one base and print the result in the base you want. To change either base back to base 10, just enter **q** followed by a carriage return.

Changing the Input Base The following example shows how to change the input base to base 16 and then add 13 to 16.

```
$ dc<CR>
16<CR>
i<CR>
13<CR>
16<CR>
+<CR>
p<CR>
65
q<CR>
$
```

Changing the Output Base The following example shows how to change the output base to base 8 and add 12 to 12.

```
$ dc<CR>
8<CR>
0<CR>
12<CR>
12<CR>
+<CR>
p<CR>
30
q<CR>
$
```

Changing the Accuracy of Calculations

You can increase the accuracy of calculations by increasing the number of digits after the decimal point. To do this in **dc** you change the scale (scale is shown as **k**). The scale in **dc** is normally set to 0. After you change the scale from 0, to get **dc** back to a scale of 0, enter **q** followed by a carriage return.

In the following example, you want to change the scale to 5, then divide 100 by 3, and change the scale back to 0.

```
$ dc<CR>
5<CR>
k<CR>
100<CR>
3<CR>
/<CR>
p<CR>
33.33333
q<CR>
$
```

dirname — Print Portions of Path Names

General

The **dirname** command is mainly used in shell scripts (programs) to output the directory name portion of a path name. It is usually enclosed in grave accents (‘command *arguments*‘) for command substitution. (Note that grave accents are also called back quotes.)

> ***Note:*** The **dirname** command is described in the manual pages under the **basename** command.

Command Format

The **dirname** command has the following format:

dirname *string*

The *string* argument is a path name. The last " /name" portion of the path name will not be printed.

Sample Command

The following command line entries and system responses show basic operation of **dirname** command. A string is specified as an argument to the **basename** command. The last portion of the path name specified is deleted.

```
$ dirname /f2/joan/letters/jack<CR>
/f2/joan/letters
$ NAME=‘dirname /f2/joan/letters/jack‘<CR>
$ echo $NAME<CR>
/f2/joan/letters
$
```

env — Set Environment for Command Execution

General

The **env** command momentarily changes the user's environment variables for execution of a command. A user's environment is automatically initialized upon login. The environment variables of the shell are unchanged. For information on the variables and how they affect the executing environment, refer to the section in the *UNIX System User Guide* where shell programming variables are described.

Command Format

The **env** command has the following format:

env *[-] [name=value] ... [command [arguments]]*

The **-** flag eliminates the current environment so that the environment consists of the specified variables only.

Arguments of the form *name=value* are environment variable(s) that are to be inherited into the current environment before executing *command arguments*. More than one argument of the form *name=value* can be listed.

If there is no *command* specified, the **env** command will print the resulting environment. This is useful for checking the environment before executing a command.

Sample Command

In a shell program, you may have a need to temporarily change the PATH variable in the executing environment. You may have several versions of a program, and each program and the files required for its execution are located in its own directory. If the current PATH variable does not include the path needed, you could use the **env** command to temporarily change the path as follows:

```
$ env - PATH=/usr/rsc/dbase/labels1.4 getlabels/fP<CR>
$
```

where **getlabels** is a shell program to be executed using the files located in the **labels1.4** directory. With the **-** flag specified, all other paths will be ignored. Therefore, all files required for execution must reside in **labels1.4**.

factor — Find Prime Factors of a Number

General

The **factor** command enables you to calculate the prime factors of any whole number.

Command Format

There are two formats for the **factor** command. These formats are:

factor *number*

factor
number

The *number* argument is the number that you want to factor. The results are the same regardless of which format you use.

Sample Calculations

The following examples show the two different ways to execute the **factor** command. 12 is the number to be factored. 2,2,3 are the prime factors of 12, with the prime factor of 1 not shown.

Factor One Number

```
$ factor 12<CR>
12
      2
      2
      3
$
```

Factor More Than One Number

When factoring more than one number, enter **factor** followed by a carriage return. Then, enter the number you want to factor followed by a carriage return. The answer will then be displayed. To factor another number, simply enter the number followed by a carriage return. The answer will then be displayed. You may continue to factor numbers as long as you like. To quit the **factor** command and return to the UNIX System, enter **q** followed by a carriage return.

```
$ factor<CR>
12<CR>
      2
      2
      3
4<CR>
      2
      2
q<CR>
$
```

line — Read One Line

General

The **line** command reads one line (up to a new-line character) from the standard input and copies it to the standard output. This is useful when you need to read a line from a file, or capture the line in a shell variable. An exit code of 1 is returned if an end-of-file is encountered and at least one new-line is printed. The **line** command is often used within shell programs to read from the user's terminal.

Command Format

The **line** command has no arguments or options.

Sample Command

The following program (**database**) gives you an example of how the **line** command can be used in shell programs. The program will continue to loop until you depress the BREAK key.

```
$ cat database<CR>
# usage:  This program is useful when receiving information
# from another person and storing it for later use.
#
while true
do
                echo "Enter Name: \c"
                name=`line`
                echo "Enter Address: \c"
                addr=`line`
                echo "Enter Phone Number: \c"
                phone=`line`
                echo "$name      $addr            $phone" >> file
done
$
```

logname — Print Login Name

General

The **logname** command displays the contents of the environment variable **$LOGNAME**. The **$LOGNAME** variable is set for a user upon login.

Command Format

The **logname** command has no arguments or options.

Sample Commands

The following example shows how to enter the **logname** command and the output that would follow if you were logged in as **root** (#).

```
# logname<CR>
root
#
```

This example shows the output of the **logname** command if you were a normal user logged in as **user5**.

```
$ logname<CR>
user5
$
```

machid commands (pdp11, u370, u3b, u3b5, vax) — Identify Type of Computer

General

The machid commands are used to identify the type of computer a program is on. They will return a true value (exit code of 0) if you are on a computer that the following command names indicate.

pdp11	True if you are on a PDP-11/45 or PDP-11/70 Computer.
u370	True if you are on an IBM 370 Computer.
u3b	True if you are on a 3B20S Computer.
u3b2	True if you are on a 3B2 Computer.
u3b5	True if you are on a 3B5 Computer.
vax	True if you are on a VAX-11/750 or VAX-11/780 Computer.

Note: The **u3b2** command is included in the Essential Utilities commands and does not need to be installed.

The commands that do not apply to your machine will return a false (nonzero) value. These commands are often used to make a shell program or makefile more portable.

Command Format

The **machid** commands have no arguments or options.

Sample Command

The following example is a simple shell program that demonstrates the use of each **machid** command. The **cat** command is used to display the contents of **userlimit**. The **userlimit** program is then executed.

```
$ cat userlimit<CR>
# usage: userlimit test to see what type of computer
# is being used and print the computer type.
# if computer is a 3B2 Computer, set the user limits
# to 1024 and then print the limits.
# if computer is any other computer, just print the
# user limits
if u3b2
then
ulimit 1024              # set user limits to 1024
echo "THIS IS A 3B2 COMPUTER"
elif u3b5
then
echo "THIS IS A 3B5 COMPUTER"
elif u3b
then
echo "THIS IS A 3B20S COMPUTER"
elif u370
then
echo "THIS IS AN IBM 370 COMPUTER"
elif pdp11
then
echo "THIS IS A PDP-11/45 OR PDP-11/70 COMPUTER"
elif vax
then
echo "THIS IS A VAX-11/750 OR VAX-11/780 COMPUTER"
fi
echo "The ulimit is \c"
A=`ulimit`
echo $A                  # echoes the value of the
                         # user limits on terminal
                         # screen
$ userlimit<CR>
THIS IS A 3B2 COMPUTER
The ulimit is 1024
$
```

The ***userlimit*** program illustrates a technique for setting the process file size limit while on a 3B2 Computer. The program checks to see if you are on a 3B2 Computer. If you are on a 3B2 Computer, the

program will let you know that you are on a 3B2 Computer. The process file size limit is then set to 1024. A final statement, printed on your terminal screen, confirms that the process file size limit was set to 1024. If you are not on a 3B2 Computer, the program will let you know what type of computer you are on and what is your process file size limit.

nice — Run a Command at Low Priority

General

The **nice** command executes a command at a lower central processing unit (CPU) priority level.

Command Format

The **nice** command has the following format:

nice *[-increment] command [arguments]*

The *increment* argument specifies the priority level to be lowered for the execution of *command*. The *increment* specified must be in the range of 1 through 19, where the larger the value the lower CPU priority. If no argument is specified for *increment*, an increment of 10 is assumed.

Sample Command

Compiling C programs can consume a large amount of CPU time and greatly reduce the overall system response time. To prevent degradation of response time, compilation commands can be assigned a lower priority level. The following example shows how to use the **nice** command in conjunction with the **cc** command.

```
$ nice -19 cc source.c&<CR>
3745
$
```

Note the (**&**) prior to the (<*CR*>). This causes the compilation to be done in the background. After the <*CR*>, the process identification number (3745) is echoed. You are then returned to the shell while the make command runs as a background process with a very low priority.

nohup — Run a Command Immune to Hangups or Quits

General

The **nohup** command allows you to execute a command that is immune to hangups and quits. The use of the ampersand (&) with **nohup** puts the command in the background and allows you to log off the system without terminating the process running under **nohup**. To interrupt a command operating under **nohup**, depress the BREAK key.

Command Format

The **nohup** command has the following format:

nohup *command [arguments]*

If the output of the command is not redirected in the command line, it is sent to a file called **nohup.out**. If the **nohup.out** file is not writable in the current directory, the output is directed to **$HOME/nohup.out**.

Sample Commands

In the following example, a **diff** command is run under **nohup** and an attempt is made to log off during the processing of the command.

```
$ nohup diff source source2<CR>
Sending output to nohup.out
exit<CR>
$
login:
```

Note that the user was not logged off until the **diff** command was completed. Since the output was not redirected to another file, the output was sent to **nohup.out**.

In this example, the output of the **diff** command is sent to a file called **differ**. The command is run in the background so that the user can log off while the processing of the **diff** command continues.

```
$ nohup diff source source2 > differ&<CR>
4025
$ exit<CR>
login:
```

Immediately after the nohupped command is entered, the background process number is displayed and the user is returned to the shell. The user can then log off the system without terminating the background process or enter another command.

shl — Layered Shell

General

The **shl** command allows a user to interact with more than one shell from a single terminal. This is done by alternating control of the terminal between several shells which are known as layers. The layer that can accept input from the keyboard is considered the " current layer."

The maximum number of layers that you can invoke with **shl** is eight. The first layer, however, is used by **shl** to manipulate the other layers. Its prompt is >>>. The other layers have a prompt corresponding to their name. Layer names can have up to eight significant characters, but cannot be the default names **1** through **7** or **(1)** through **(7)**. The default names are assigned in order as layers are created.

Some examples of what you might want to do in layers would be:

Layer 1 Edit a file.

Layer 2 Use as interactive shell.

Layer 3 Execute a text processing command such as " nroff" .

Layer 4 Edit another file.

There are several commands which may be issued from the shl prompt level. These commands allow you to create layers, delete layers, switch between layers, and manipulate the output from each layer. The commands can be abbreviated by any prefix of the command (for example: **c** instead of **create**.) The following list describes the use of each command:

create *[name]*
Create a layer called *[name]* and make it the current layer. If no name is given, a default name is used.

block *name [name...]*
For each *[name]*, block the output of the corresponding layer when it is not the current layer.

delete *name [name...]*
For each *[name]*, delete the corresponding layer.

help or **?**
Print the syntax of the shl commands.

layers *[-l] [name...]*
For each *[name]* list the layer name and its process group id. If no name is given, information is given for all processes. The **-l** option gives a more detailed listing.

resume *[name]*
Make the referenced layer the current layer.

toggle
Resume the layer that was previously current.

unblock *name [name...]*
For each *[name]*, do not block the output of the corresponding layer when it is not the current layer.

quit Exit shl.

name Make the layer referenced by *name* the current layer. If the name is the same as any prefix of a shl command, the "**resume** *name*" command must be used.

When you are in a layer and wish to return to the **shl**, enter a <CTRL **z**>.

Command Format

The **shl** command has no arguments or options.

Sample Command

The following example shows how to enter the **shl** command, create two more layers, execute some commands in the layers, and then return to the UNIX System.

```
$ shl<CR>
>>> create<CR>
(1) pwd<CR>
/usr/abc
(1) ls<CR>
file.c
memo1
(1) nroff -cm memo1 > formatmemo<CR>
<CTRL z>  >>> c Layer2<CR>
Layer2 pwd<CR>
/usr/abc
Layer2 ed file.c<CR>
36
1,$p<CR>
main()
{
        printf(''hello wold/n'');
}
<CTRL z>  >>> layers -l<CR>
(1) (3650) executing or awaiting input
     UID   PID  PPID  C    STIME TTY     TIME COMMAND
     abc  3669  3650  0 14:24:52 sxt001  0:01 nroff -cm memo1
     abc  3650  3649  0 14:23:50 sxt001  0:00 /bin/sh -i
Layer2 (3700) executing or awaiting input
     UID   PID  PPID  C    STIME TTY     TIME COMMAND
     abc  3705  3690  0 14:25:56 sxt002  0:01 ed file.c
     abc  3690  3649  0 14:25:02 sxt002  0:00 /bin/sh -i
>>> r<CR>
resuming Layer2
3s/wold/world/p<CR>
        printf(''hello world0');
w<CR>
37
q<CR>
Layer2 <CTRL z>  >>> layers -l 1<CR>
(1) (3650) executing or awaiting input
     UID   PID  PPID  C    STIME TTY     TIME COMMAND
     abc  3650  3649  0 14:23:50 sxt001  0:00 /bin/sh -i

>>> q<CR>
$
```

Note: User knows nroff has completed because it is not listed.

tabs — Set Tab Stops on a Printer or Terminal

General

The **tabs** command is used to set tabs on a printer or terminal. Note that the printer must be enabled to set tabs. Also note that not all terminals are compatible with the escape sequences output by the tabs command to clear existing tabs and set new tabs. Executing the command without arguments sets tabs every 8 spaces. Tab stops set every 8 spaces is the standard default setting used by most programs (commands) for high-speed output. Arguments can be provided to the tabs command that specify particular tab stops. Tabs specifications can also be saved in a file.

Command Format

The **tabs** command has the following format:

tabs *[tab specification] [+***m***n] [-***T***type]*

The [*tab specification*] argument specifies the placement of tabs.

The +**m***n* argument specifies the position of the left margin, where *n* is the number of spaces to be indented. Omitting the *n* from the argument causes a default indent of 10 spaces.

The -**T***type* argument specifies the terminal type. Both the margin and terminal type arguments can be omitted from a command line for most applications.

Canned Tab Specifications

A variety of ready-made tab specifications can be called by the **tabs** command by specifying the appropriate option code. Refer to the **tabs** manual pages in the Appendix for identification of the various codes.

Sample Commands

The following examples show different types of **tabs** command specifications. The following command line sets tab stops for use in writing FORTRAN programs.

```
$ tabs -f<CR>
$
```

A repetitive tab specification sets tab stops every " n" spaces. The following command sets tab stops every 6 spaces.

```
$ tabs -6<CR>
$
```

Tabs can be set at selected points. The following command line sets tabs at 5, 20, 23, and 40 spaces on a line.

```
$ tabs 5,20,23,40<CR>
$
```

A specification line that sets tabs at 20, 25, and 32 is as follows:

```
<:t20,25,32:>
```

time — Time a Command

General

The **time** command determines how long it takes the specified command to execute. After the command has finished execution, the following times are reported in seconds:

real	time that elapsed during execution
user	time spent executing user code
sys	time spent executing system code.

Command Format

The **time** command has the following format:

time *command [arguments]*

The **time** command does not have any arguments other than the actual command you wish to measure.

Sample Command

The following example shows how to enter the **time** command and the response that follows:

```
$ time id<CR>
uid=101(rsc) gid=1(other)

real                    1.2
user                    0.0
sys                     0.2
$
```

tty — Print the Terminal Name

General

The **tty** command prints the pathname of the user's terminal.

Command Format

The **tty** command has the following format:

tty [*-l*] [*-s*]

The *-l* option prints the synchronous line number to which the user's terminal is connected (if connected to a synchronous line).

The *-s* option inhibits printing of the terminal pathname allowing the user to test the exit code.

The exit codes for the **tty** command are different from those of other commands. The exit codes for the **tty** command are as follows:

0	Standard input is a terminal
1	Standard input is from source other than terminal
2	Invalid option(s) were specified.

Sample Commands

The following examples show how to enter the **tty** command and the response that would follow:

```
$ tty<CR>
/dev/tty21
$
```

The *-s* option is used to check the exit code as follows:

```
$ tty -s<CR>
$ echo $?<CR>
0
$
```

The question mark (**?**) is the special shell variable, representing the exit code.

units — Find Unit Conversion Factors

General

The **units** command enables you to find the conversion factors (divisor and multiplier) for converting from one unit of measurement to another related unit of measurement. The types of unit measurements which can be converted include: length, weight, mass, electrical, money, liquid, etc. The file **/usr/lib/unittab** contains a list of units you can convert using the **units** command.

Command Format

The **units** command has the following format:

units
you have: *argument*
you want: *argument*

The **you have:** *argument* identifies the unit of measure that you are using as a reference.

The **you want:** *argument* identifies the unit of measure you want to convert to.

The response will be two different numbers. The first number will be a multiplier that you use as a conversion factor. The second number will be a divisor that can also be used as a conversion factor.

Sample Calculations

You must have a system prompt (**$**) to begin the use of the **units** command. To execute a **units** command, simply enter **units** followed by a carriage return. The system will respond with the display "`you have:`". Enter the unit of measurement you have, followed by a carriage return. The system will then display "`you want:`". Now, enter the unit of measurement that you want, followed by a carriage return. The system will display two

conversion factors: one will be a multiplier (shown with a * symbol), the other will be a divisor (shown with a / symbol). The system will also display "you have:", so you can continue using the **units** command. To quit the **units** command, press the <DEL> or <BREAK> key. The system prompt (**$**) will then be displayed to indicate that you have returned to the UNIX System.

In this example, inch is the unit of measurement you have, and mile is the unit of measurement you want. The answer *1.578283e-05 means to multiply inch by 1.578283 times 10 raised to the -5 power (0.00001578283) to get miles. The answer /6.336000e+04 means to divide inch by 6.336000 times 10 raised to the +4 power (6336.00) to get miles.

```
$ units<CR>
you have: inch<CR>
you want: mile<CR>
        * 1.578283e-05
        / 6.336000e+04
you have: <BREAK>
$
```

If you enter an invalid unit after the system displays "you want:", the system will respond with "cannot recognize". For example,

```
$ units<CR>
you have: inch<CR>
you want: frt<CR>
cannot recognize frt
you want:  <BREAK>
$
```

To correct this error, reenter the correct unit after "you want:" is displayed again.

If you try to find the conversion factors for unrelated unit measurements, the system will respond with the message "**conformability**" and conversion factors to convert the entered units to related units. This message means the unit conversions you want cannot be done. For example,

```
$ units<CR>
you have: inch<CR>
you want: lbs<CR>
conformability
2.540000e-02m
4.535924e-01kg
you have: <BREAK>
$
```

The response 2.540000e-02m means multiplying an inch by 0.0254 will give you meters (m). The response 4.535924e-01kg means multiplying lbs by 0.4535924 will give you kilograms (kg).

xargs — Construct Argument List(s) and Execute Command

General

The **xargs** command is used to execute a command or a shell program one or more times by combining arguments to the **xargs** command with arguments read from the standard input. Every time a command or a shell program is invoked through the **xargs** command, the number of arguments read and the manner in which the arguments are combined are determined by the flags specified.

Command Format

The **xargs** command has the following format:

xargs *[flags] [command [initial-arguments]]*

The *flag* values are as follows:

-l*[number]*
: *Command* is executed after every *[number]* of lines is read in. Fewer lines of arguments will be used when the last invocation of *command* occurs and there are fewer than *[number]* before the end of the file. A line ends with the first new-line character, unless the last character of the line is a blank or a tab. In this case, a line continues onto the next line. If *number* is not specified, " 1" is assumed. Option **-x** is automatically used when you use this option.

-i*[replstr]*
: This is the insert mode. *Command* is executed for each line. Each line is considered a single argument and substituted into each occurrence of *[replstr]* in the initial arguments. No more than five initial arguments may contain *[replstr]*. The *[replstr]* can occur more than five times if it occurs more than once in a single argument. Blanks and tabs at the beginning of each line are thrown away. Constructed arguments may not grow larger than 255 characters. Option **-x** is automatically used when you

use this option. {} is assumed for *[replstr]* if it is not specified.

-n*[number]*
Command is executed once for every *[number]* of arguments read in. Fewer arguments will be used for the invocation of a command if their total size is greater than *size* characters (see **-s***size* option). Fewer arguments will be used for the last command invocation if there are fewer than " *number*" arguments before the end of the file. If option **-x** is also used, each " *number*" arguments must not exceed the *size* limitation or **xargs** will terminate execution.

-t This is the trace mode. The *command* and each constructed argument list are echoed to file descriptor 2 just prior to their execution.

-p This is the prompt mode. You are asked if you want to execute *command* before each invocation. The trace mode (**-t**) is automatically turned on to print *command* to be invoked. A **?...** prompt will follow *command*. A reply of **y** (optionally followed by anything) will execute *command*. Any other response, including a carriage return, will skip that particular invocation of *command*.

-x Causes the **xargs** command to terminate if any argument list would be greater than *size* characters. This option is used automatically if options **-i** or **-l** are used. The total length of all arguments must be within the *size* limit if options **-i, -l**, or **-n** are not specified.

-s*[size]*
The maximum size of each argument list is set to *size* characters. *Size* must be a positive integer less than or equal to 470. If this option is not specified, 470 is taken as the default. The character count for *size* includes one extra character for each argument and the count of characters in the command name.

-e*[eofstr]*
The *eofstr* is taken as the logical end-of-file string. Underbar (_) is assumed for the logical end-of-file string if

this option is not specified. If you use this option with no *eofstr*, the logical end-of-file string capability is turned off (_ is taken literally). The **xargs** command will read the standard input until the end of the file is reached or the logical end-of-file string is encountered.

Command, which may be a shell program, is searched using your **variable PATH**. If *command* is omitted, **/bin/echo** is used.

Arguments read from the standard input are defined to be continuous strings of characters. These strings of characters are delimited by one or more blanks, tabs, or new-lines. Empty lines are always discarded. Blanks and tabs may be embedded as part of an argument if escaped or quoted. To escape the next character, precede that character with a backslash (\). Characters enclosed in quotes (single or double) are taken literally [including backslash (\)] and the delimiting quotes are removed.

Each argument list is constructed of *initial-arguments* followed by some number of arguments read from the standard input, except when the **-i** flag is used. When the **-i, -l**, and **-n** flags are not specified, the *initial-arguments* are followed by arguments read continuously from the standard input. These arguments will continue to be read until an internal buffer is full. After the buffer is full, *command* is executed with the accumulated arguments. When there are conflicts between flags, such as using **-l** and **-n** together, the last flag specified has precedence.

The **xargs** command will terminate if it receives a return code of **-1**. It will also terminate if it cannot execute *command*. When *command* is a shell program, **it** should *exit* explicitly with an appropriate value to avoid accidentally returning with a return code of **-1**.

Sample Commands

This example shows how the **xargs** command works when using the **-l** option and designating 5 lines. Since no output argument is given, the output is echoed onto the display.

```
$ xargs -15<CR>
11111<CR>
222<CR>
33<CR>
4444<CR>
5<CR>
11111 222 33 4444 5
AAAA<CR>
BB<CR>
<CTRL d>
AAAA BB
$
```

This example shows how the **xargs** command works when using the **-i** option. The maximum number of arguments (5) is used.

```
$ xargs -i echo {}a{} b{} c{}{} d{} {}e<CR>
x<CR>
xax bx cxx dx xe
z z<CR>
z zaz z bz z cz zz z dz z z ze
<CTRL d>
$
```

This example shows how to use the **xargs** command to move files to different directories. For each file in dir1, you are asked if you want to move that file to dir2. If you respond with a **y**, the file is moved to dir2. If you give any other response, the file is left in dir1.

```
$ ls dir1 | xargs -i -p mv dir1/{} dir2/{}<CR>
mv dir1/file1 dir2/file1 ?...y<CR>
mv dir1/file2 dir2/file2 ?...y<CR>
mv dir1/file3 dir2/file3 ?...n<CR>
mv dir1/file4 dir2/file4 ?...y<CR>
$ ls dir1<CR>
file3
$ ls dir2<CR>
file1
file2
file4
$
```

The **filetest** program illustrates a simple use of the **xargs** command. The first argument (**$1**) is checked to see if it is a file or a directory. If **$1** is a file, the file contents are printed on your terminal screen. If **$1** is a directory, you are asked if you want to move the files in that directory, one at a time, to the specified directory (**$2**). If **$1** is neither a file nor a directory, a message is printed indicating that **$1** is neither a file nor a directory.

```
$ cat filetest<CR>
# usage: filetest testfile directory
if test -f "$1"          # is $1 a file?
then
cat $1
elif test -d "$1"        # else, is $1 a directory?
then
ls $1 | xargs -i -p mv $1/{} $2/{}
                         # move files from $1 to $2
                         # if response is y
else
echo $1 is neither a file nor a directory
fi
$ ls<CR>
file1
file2
test1
test2
$ filetest test1 test2<CR>
mv test1/cec1 test2/cec1 ?...y<CR>
mv test1/cec2 test2/cec2 ?...n<CR>
mv test1/cec3 test2/cec3 ?...n<CR>
mv test1/cec4 test2/cec4 ?...y<CR>
$ ls test1<CR>
cec2
cec3
$ ls test2<CR>
cec1
cec4
$
```

Appendix

MANUAL PAGES

This appendix contains the UNIX System Manual Pages for the User Environment Utilities. Manual pages for the following commands are provided in alphabetical sequence.

at	**crontab**	**nice**	**u370**
banner	**dc**	**nohup**	**u3b**
basename	**dirname**	**pdp11**	**u3b5**
batch	**env**	**shl**	**units**
bc	**factor**	**tabs**	**vax**
cal	**line**	**time**	**xargs**
calendar	**logname**	**tty**	

Certain of these commands are described on a manual page along with one or more other commands. Commands that are described on a different manual page than indicated by the name of the command are as follows.

COMMAND	MANUAL PAGE
batch	at
dirname	basename
pdp11	machid
u370	machid
u3b	machid
u3b5	machid
vax	machid

You have the capability to arrange the manual pages provided in this guide to support your local needs. The yellow sheet provided in this manual describes your options for filing the manual pages as well as the descriptive information.

For your convenience, the user manual pages for the User Environment Utilities are provided in both this guide and alphabetically in the *3B2 User Reference Manual.*

NAME

at, batch — execute commands at a later time

SYNOPSIS

at *time* [*date*] [+ *increment*]
at −*r* job...
at -*l*[*job...*]

batch

DESCRIPTION

At and *batch* read commands from standard input to be executed at a later time. The *sh*(1) utility provides different way of specifying standard input. *At* allows you to specify when the commands should be executed, while jobs queued with *batch* will execute when system load level permits. *At* may be used with the following options:

−r Removes jobs previously scheduled with *at*.

−l Reports all jobs scheduled for the invoking user.

Standard output and standard error output are mailed to the user unless they are redirected elsewhere. The shell environment variables, current directory, umask, and ulimit are retained when the commands are executed. Open file descriptors, traps, and priority are lost.

Users are permitted to use *at* if their name appears in the file **/usr/lib/cron/at.allow**. If that file does not exist, the file **/usr/lib/cron/at.deny** is checked to determine if the user should be denied access to *at*. If neither file exists, only root is allowed to submit a job. If **at.deny** is empty, global usage is permitted. The allow/deny files consist of one user name per line.

The *time* may be specified as 1, 2, or 4 digits. One and two digit numbers are taken to be hours, four digits to be hours and minutes. The time may alternately be specified as two numbers separated by a colon, meaning *hour:minute*. A suffix **am** or **pm** may be appended; otherwise a 24-hour clock time is understood. The suffix **zulu** may be used to indicate GMT. The special names **noon**, **midnight**, **now**, and **next** are also recognized.

An optional *date* may be specified as either a month name followed by a day number (and possibly year number preceded by an optional comma) or a day of the week (fully spelled or abbreviated to three characters). Two special "days", **today** and **tomorrow** are recognized. If no *date* is given, **today** is assumed if the given hour is greater than the current hour and **tomorrow** is assumed if it is less. If the given month is less than the current month (and no year is given), next year is assumed.

The optional *increment* is simply a number suffixed by one of the following: **minutes**, **hours**, **days**, **weeks**, **months**, or **years**. (The singular form is also accepted.)

Thus legitimate commands include:

```
at 0815am Jan 24
at 8:15am Jan 24
at now + 1 day
at 5 pm Friday
```

At and *batch* write the job number and schedule time to standard error.

Batch submits a batch job. It is almost equivalent to "at now", but not quite. For one, it goes into a different queue. For another, "at now" will respond with the error message **too late**.

At **-r** removes jobs previously scheduled by *at* or *batch*. The job number is the number given to you previously by the *at* or *batch* command. You can also get job numbers by typing *at* **-l**. You can only remove your own jobs unless you are the super-user.

EXAMPLES

The *at* and *batch* commands read from standard input the commands to be executed at a later time. *Sh*(1) provides different ways of specifying standard input. Within your commands, it may be useful to redirect standard output.

This sequence can be used at a terminal:

```
batch
sort filename >outfile
<control-D> (hold down 'control' and depress 'D')
```

This sequence, which demonstrates redirecting standard error to a pipe, is useful in a shell procedure (the sequence of output redirection specifications is significant):

```
batch <<!
sort filename 2>&1 >outfile | mail loginid
!
```

To have a job reschedule itself, invoke *at* from within the shell procedure, by including code similar to the following within the shell file:

```
echo "sh shellfile" | at 1900 thursday next week
```

FILES

/usr/lib/cron	main cron directory
/usr/lib/cron/at.allow	list of allowed users
/usr/lib/cron/at.deny	list of denied users
/usr/lib/cron/queue	scheduling information
/usr/spool/cron/atjobs	spool area

SEE ALSO

kill(1), mail(1), nice(1), ps(1), sh(1), sort(1).
cron(1M) in the *3B2 Computer System Administration Utilities Guide.*

DIAGNOSTICS

Complains about various syntax errors and times out of range.

NAME

banner — make posters

SYNOPSIS

banner strings

DESCRIPTION

Banner prints its arguments (each up to 10 characters long) in large letters on the standard output.

SEE ALSO

echo(1).

NAME

basename, dirname — deliver portions of path names

SYNOPSIS

basename string [suffix]
dirname string

DESCRIPTION

Basename deletes any prefix ending in / and the *suffix* (if present in *string*) from *string*, and prints the result on the standard output. It is normally used inside substitution marks (` `) within shell procedures.

Dirname delivers all but the last level of the path name in *string*.

EXAMPLES

The following example, invoked with the argument **/usr/src/cmd/cat.c**, compiles the named file and moves the output to a file named **cat** in the current directory:

```
cc $1
mv a.out `basename $1 '\.c'`
```

The following example will set the shell variable NAME to **/usr/src/cmd**:

```
NAME=`dirname /usr/src/cmd/cat.c`
```

SEE ALSO

sh(1).

BUGS

The *basename* of / is null and is considered an error.

NAME

bc — arbitrary-precision arithmetic language

SYNOPSIS

bc [**−c**] [**−l**] [file ...]

DESCRIPTION

Bc is an interactive processor for a language that resembles C but provides unlimited precision arithmetic. It takes input from any files given, then reads the standard input. The *bc*(1) utility is acutally a preprocessor for *dc*(1), which it invokes automatically unless the **−c** option is present. In this case the *dc* input is sent to the standard output instead. The options are as follows:

−c Compile only. The output is send to the standard output.

−l Argument stands for the name of an arbitrary precision math library.

The syntax for *bc* programs is as follows; L means letter a−z, E means expression, S means statement.

Comments
- are enclosed in /* and */.

Names
- simple variables: L
- array elements: L [E]
- The words "ibase", "obase", and "scale"

Other operands
- arbitrarily long numbers with optional sign and decimal point.
- (E)
- sqrt (E)
- length (E) number of significant decimal digits
- scale (E) number of digits right of decimal point
- L (E , ... , E)

Operators
- + − * / % ^ (% is remainder; ^ is power)
- ++ −− (prefix and postfix; apply to names)
- == <= >= != < >
- = =+ =− =* =/ =% =^

Statements
- E
- { S ; ... ; S }
- if (E) S
- while (E) S
- for (E ; E ; E) S
- null statement
- break
- quit

Function definitions

```
define L ( L ,..., L ) {
        auto L, ... , L
        S; ... S
        return ( E )
}
```

Functions in **−l** math library

s(x)	sine
c(x)	cosine
e(x)	exponential
l(x)	log

a(x)	arctangent
j(n,x)	Bessel function

All function arguments are passed by value.

The value of a statement that is an expression is printed unless the main operator is an assignment. Either semicolons or new-lines may separate statements. Assignment to *scale* influences the number of digits to be retained on arithmetic operations in the manner of *dc*(1). Assignments to *ibase* or *obase* set the input and output number radix respectively.

The same letter may be used as an array, a function, and a simple variable simultaneously. All variables are global to the program. "Auto" variables are pushed down during function calls. When using arrays as function arguments or defining them as automatic variables, empty square brackets must follow the array name.

EXAMPLE

```
scale = 20
define e(x){
        auto a, b, c, i, s
        a = 1
        b = 1
        s = 1
        for(i=1; 1==1; i++){
                a = a*x
                b = b*i
                c = a/b
                if(c == 0) return(s)
                s = s+c
        }
}
```

defines a function to compute an approximate value of the exponential function and

```
for(i=1; i<=10; i++) e(i)
```

prints approximate values of the exponential function of the first ten integers.

FILES

/usr/lib/lib.b	mathematical library
/usr/bin/dc	desk calculator proper

SEE ALSO

dc(1).

BUGS

No && , || yet.
For statement must have all three expressions (E's).
Quit is interpreted when read, not when executed.

NAME

cal — print calendar

SYNOPSIS

cal [[month] year]

DESCRIPTION

Cal prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. If neither is specified, a calendar for the present month is printed. *Year* can be between 1 and 9999. The *month* is a number between 1 and 12. The calendar produced is that for England and her colonies.

Try September 1752.

BUGS

The year is always considered to start in January even though this is historically naive.
Beware that "cal 83" refers to the early Christian era, not the 20th century.

NAME

calendar — reminder service

SYNOPSIS

calendar [—]

DESCRIPTION

Calendar consults the file **calendar** in the current directory and prints out lines that contain today's or tomorrow's date anywhere in the line. Most reasonable month-day dates such as "Aug. 24," "august 24," "8/24," etc., are recognized, but not "24 August" or "24/8". On weekends "tomorrow" extends through Monday.

When an argument is present, *calendar* does its job for every user who has a file **calendar** in the login directory and sends them any positive results by *mail*(1). Normally this is done daily by facilities in the UNIX operating system.

FILES

/usr/lib/calprog	to figure out today's and tomorrow's dates
/etc/passwd	
/tmp/cal*	

SEE ALSO

mail(1).

BUGS

Your calendar must be public information for you to get reminder service. *Calendar's* extended idea of "tomorrow" does not account for holidays.

NAME

crontab — user crontab file

SYNOPSIS

crontab [file]
crontab -r
crontab -l

DESCRIPTION

Crontab copies the specified file, or standard input if no file is specified, into a directory that holds all users' crontabs. The −r option removes a user's crontab from the crontab directory. *Crontab* −l will list the crontab file for the invoking user.

Users are permitted to use *crontab* if their names appear in the file **/usr/lib/cron/cron.allow.** If that file does not exist, the file **/usr/lib/cron/cron.deny** is checked to determine if the user should be denied access to *crontab.* If neither file exists, only root is allowed to submit a job. If either file is at **cron.deny**, global usage is permitted. The allow/deny files consist of one user name per line.

A crontab file consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns that specify the following:

minute (0−59),
hour (0−23),
day of the month (1−31),
month of the year (1−12),
day of the week (0−6 with 0=Sunday).

Each of these patterns may be either an asterisk (meaning all legal values) or a list of elements separated by commas. An element is either a number or two numbers separated by a minus sign (meaning an inclusive range). Note that the specification of days may be made by two fields (day of the month and day of the week). If both are specified as a list of elements, both are adhered to. For example, 0 0 1,15 * 1 would run a command on the first and fifteenth of each month, as well as on every Monday. To specify days by only one field, the other field should be set to * (for example, 0 0 * * 1 would run a command only on Mondays).

The sixth field of a line in a crontab file is a string that is executed by the shell at the specified times. A percent character in this field (unless escaped by \) is translated to a new-line character. Only the first line (up to a % or end of line) of the command field is executed by the shell. The other lines are made available to the command as standard input.

The shell is invoked from your **$HOME** directory with an **arg0** of **sh.** Users who desire to have their *.profile* executed must explicitly do so in the crontab file. *Cron* supplies a default environment for every shell, defining **HOME, LOGNAME, SHELL(=/bin/sh)**, and **PATH(=:/bin:/usr/bin:/usr/lbin)**.

NOTE: Users should remember to redirect the standard output and standard error of their commands! If this is not done, any generated output or errors will be mailed to the user.

FILES

/usr/lib/cron	main cron directory
/usr/spool/cron/crontabs	spool area
/usr/lib/cron/log	accounting information
/usr/lib/cron/cron.allow	list of allowed users
/usr/lib/cron/cron.deny	list of denied users

SEE ALSO

sh(1).
cron(1M) in the *3B2 Computer System Administration Utilities Guide.*

NAME

dc — desk calculator

SYNOPSIS

dc [file]

DESCRIPTION

Dc is an arbitrary precision arithmetic package. Ordinarily it operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained. (See *bc*(1), a preprocessor for *dc* that provides infix notation and a C-like syntax that implements functions. *Bc* also provides reasonable control structures for programs.) The overall structure of *dc* is a stacking (reverse Polish) calculator. If an argument is given, input is taken from that file until its end, then from the standard input. The following constructions are recognized:

number
: The value of the number is pushed on the stack. A number is an unbroken string of the digits 0—9. It may be preceded by an underscore (_) to input a negative number. Numbers may contain decimal points.

+ − / * % ^
: The top two values on the stack are added (+), subtracted (−), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. Any fractional part of an exponent is ignored.

s*x*
: The top of the stack is popped and stored into a register named *x*, where *x* may be any character. If the **s** is capitalized, *x* is treated as a stack and the value is pushed on it.

l*x*
: The value in register *x* is pushed on the stack. The register *x* is not altered. All registers start with zero value. If the **l** is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

d
: The top value on the stack is duplicated.

p
: The top value on the stack is printed. The top value remains unchanged. **P** interprets the top of the stack as an ASCII string, removes it, and prints it.

f
: All values on the stack are printed.

q
: exits the program. If executing a string, the recursion level is popped by two. If **q** is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

x
: treats the top element of the stack as a character string and executes it as a string of *dc* commands.

X
: replaces the number on the top of the stack with its scale factor.

[...]
: puts the bracketed ASCII string onto the top of the stack.

<*x* **>***x* **=***x*
: The top two elements of the stack are popped and compared. Register *x* is evaluated if they obey the stated relation.

v
: replaces the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.

!
: interprets the rest of the line as a UNIX system command.

c All values on the stack are popped.

i The top value on the stack is popped and used as the number radix for further input. **I** pushes the input base on the top of the stack.

o The top value on the stack is popped and used as the number radix for further output.

O pushes the output base on the top of the stack.

k the top of the stack is popped, and that value is used as a non-negative scale factor: the appropriate number of places are printed on output, and maintained during multiplication, division, and exponentiation. The interaction of scale factor, input base, and output base will be reasonable if all are changed together.

z The stack level is pushed onto the stack.

Z replaces the number on the top of the stack with its length.

? A line of input is taken from the input source (usually the terminal) and executed.

; : are used by *bc* for array operations.

EXAMPLE

This example prints the first ten values of n!:

```
[la1+dsa*pla10>y]sy
0sa1
lyx
```

SEE ALSO

bc(1).

DIAGNOSTICS

x is unimplemented
where *x* is an octal number.

stack empty
for not enough elements on the stack to do what was asked.

Out of space
when the free list is exhausted (too many digits).

Out of headers
for too many numbers being kept around.

Out of pushdown
for too many items on the stack.

Nesting Depth
for too many levels of nested execution.

NAME

env – set environment for command execution

SYNOPSIS

env [–] [name=value] ... [command args]

DESCRIPTION

Env obtains the current *environment*, modifies it according to its arguments, then executes the command with the modified environment. Arguments of the form *name=value* are merged into the inherited environment before the command is executed. The – flag causes the inherited environment to be ignored completely, so that the command is executed with exactly the environment specified by the arguments.

If no command is specified, the resulting environment is printed, one name-value pair per line.

SEE ALSO

sh(1).
exec(2), environ(5), profile(4) in the *3B2 Computer System Programmer Reference Manual.*

NAME

factor — factor a number

SYNOPSIS

factor [number]

DESCRIPTION

When *factor* is invoked without an argument, it waits for a number to be typed in. If you type in a positive number less than 2^{56} (about 1.0×10^{14}) it will factor the number and print its prime factors; each one is printed the proper number of times. Then it waits for another number. It exits if it encounters a zero or any non-numeric character.

If *factor* is invoked with an argument, it factors the number as above and then exits.

Maximum time to factor is proportional to $\sqrt{n}$ and occurs when n is prime or the square of a prime.

DIAGNOSTICS

"Ouch" for input out of range or for garbage input.

NAME

line — read one line

SYNOPSIS

line

DESCRIPTION

Line copies one line (up to a new-line) from the standard input and writes it on the standard output. It returns an exit code of 1 on **EOF** and always prints at least a new-line. It is often used within shell files to read from the user's terminal.

SEE ALSO

sh(1).
read(2) in the *3B2 Computer System Programmer Reference Manual.*

NAME

logname — get login name

SYNOPSIS

logname

DESCRIPTION

Logname returns the contents of the environment variable **$LOGNAME**, which is set when a user logs into the system.

FILES

/etc/profile

SEE ALSO

env(1), login(1).
environ(5), logname(3X) in the *3B2 Computer System Programmer Reference Manual.*

NAME

pdp11, u3b, u3b2, u3b5, vax — provide truth value about your processor type

SYNOPSIS

pdp11

u3b

u3b2

u3b5

vax

DESCRIPTION

The following commands will return a true value (exit code of 0) if you are on a processor that the command name indicates.

pdp11	True if you are on a PDP-11/45 or PDP-11/70.
u3b	True if you are on a 3B20 computer.
u3b2	True if you are on a 3B2 computer.
u3b5	True if you are on a 3B5 computer.
vax	True if you are on a VAX-11/750 or VAX-11/780.

The commands that do not apply will return a false (non-zero) value. These commands are often used within *make*(1) makefiles and shell procedures to increase portability.

SEE ALSO

sh(1), test(1), true(1).
make(1) in the *3B2 Computer System Extended Software Generation System Utilities.*

NAME

nice — run a command at low priority

SYNOPSIS

nice [−increment] command [arguments]

DESCRIPTION

Nice executes *command* with a lower CPU scheduling priority. If the *increment* argument (in the range 1-19) is given, it is used; if not, an increment of 10 is assumed.

The super-user may run commands with priority higher than normal by using a negative increment, e.g., **− −10**.

SEE ALSO

nohup(1).
nice(2) in the *3B2 Computer System Programmer Reference Manual.*

DIAGNOSTICS

Nice returns the exit status of the subject command.

BUGS

An *increment* larger than 19 is equivalent to 19.

NAME

nohup — run a command immune to hangups and quits

SYNOPSIS

nohup command [arguments]

DESCRIPTION

Nohup executes *command* with hangups and quits ignored. If output is not re-directed by the user, both standard output and standard error are sent to **nohup.out**. If **nohup.out** is not writable in the current directory, output is redirected to **$HOME/nohup.out**.

EXAMPLE

It is frequently desirable to apply *nohup* to pipelines or lists of commands. This can be done only by placing pipelines and command lists in a single file, called a shell procedure. One can then issue:

```
nohup sh file
```

and the *nohup* applies to everything in *file*. If the shell procedure *file* is to be executed often, then the need to type *sh* can be eliminated by giving *file* execute permission. Add an ampersand and the contents of *file* are run in the background with interrupts also ignored (see *sh*(1)):

```
nohup file &
```

An example of what the contents of *file* could be is:

```
sort ofile > nfile
```

SEE ALSO

chmod(1), nice(1), sh(1).
signal(2) in the *3B2 Computer System Programmer Reference Manual.*

WARNINGS

nohup command1; command2 *nohup* applies only to *command1*
nohup (command1; command2) is syntactically incorrect.

Be careful of where standard error is redirected. The following command may put error messages on tape, making it unreadable:

```
nohup cpio -o <list >/dev/rmt/1m&
```

while

```
nohup cpio -o <list >/dev/rmt/1m 2>errors&
```

puts the error messages into file *errors*.

NAME

shl — shell layer manager

SYNOPSIS

shl

DESCRIPTION

Shl allows a user to interact with more than one shell from a single terminal. The user controls these shells, known as *layers*, using the commands described below.

The *current layer* is the layer which can receive input from the keyboard. Other layers attempting to read from the keyboard are blocked. Output from multiple layers is multiplexed onto the terminal. To have the output of a layer blocked when it is not current, the *stty* option **loblk** may be set within the layer.

The *stty* character **swtch** (set to ^Z if NUL) is used to switch control to *shl* from a layer. *Shl* has its own prompt, >>>, to help distinguish it from a layer.

A *layer* is a shell which has been bound to a virtual tty device (**/dev/sxt???**). The virtual device can be manipulated like a real tty device using *stty*(1) and *ioctl*(2). Each layer has its own process group id.

Definitions

A *name* is a sequence of characters delimited by a blank, tab or new-line. Only the first eight characters are significant. The *names* **(1)** through **(7)** cannot be used when creating a layer. They are used by *shl* when no name is supplied. They may be abbreviated to just the digit.

Commands

The following commands may be issued from the *shl* prompt level. Any unique prefix is accepted.

create [*name*]
: Create a layer called *name* and make it the current layer. If no argument is given, a layer will be created with a name of the form *(#)* where # is the last digit of the virtual device bound to the layer. The shell prompt variable **PS1** is set to the name of the layer followed by a space. A maximum of seven layers can be created.

block *name* [*name* ...]
: For each *name*, block the output of the corresponding layer when it is not the current layer. This is equivalent to setting the *stty* option **−loblk** within the layer.

delete *name* [*name* ...]
: For each *name*, delete the corresponding layer. All processes in the process group of the layer are sent the SIGHUP signal (see *signal*(2)).

help (or **?**)
: Print the syntax of the *shl* commands.

layers [**−l**] [*name* ...]
: For each *name*, list the layer name and its process group. The **−l** option produces a *ps*(1)-like listing. If no arguments are given, information is presented for all existing layers.

resume [*name*]
: Make the layer referenced by *name* the current layer. If no argument is given, the last existing current layer will be resumed.

toggle Resume the layer that was current before the last current layer.

unblock *name* [*name* ...]
: For each *name*, do not block the output of the corresponding layer when it is not the current layer. This is equivalent to setting the *stty*

option **loblk** within the layer.

quit Exit *shl*. All layers are sent the SIGHUP signal.

name Make the layer referenced by *name* the current layer.

FILES

/dev/sxt???	Virtual tty devices
$SHELL	Variable containing path name of the shell to use (default is /bin/sh).

SEE ALSO

sh(1), stty(1).

ioctl(2), signal(2) in the *3B2 Computer System Programmer Reference Manual.*

sxt(7) in the *#b2 Computer System Administration Utilities Guide.*

NAME

tabs – set tabs on a terminal

SYNOPSIS

tabs [tabspec] [**+mn**] [**–T**type]

DESCRIPTION

Tabs sets the tab stops on the user's terminal according to the tab specification *tabspec*, after clearing any previous settings. The user's terminal must have remotely-settable hardware tabs.

Users of GE TermiNet terminals should be aware that they behave in a different way than most other terminals for some tab settings. The first number in a list of tab settings becomes the *left margin* on a TermiNet terminal. Thus, any list of tab numbers whose first element is other than 1 causes a margin to be left on a TermiNet, but not on other terminals. A tab list beginning with 1 causes the same effect regardless of terminal type. It is possible to set a left margin on some other terminals, although in a different way (see below).

Four types of tab specification are accepted for *tabspec*: "canned," repetitive, arbitrary, and file. If no *tabspec* is given, the default value is **–8**, i.e., UNIX system "standard" tabs. The lowest column number is 1. Note that for *tabs*, column 1 always refers to the leftmost column on a terminal, even one whose column markers begin at 0, e.g., the DASI 300, DASI 300s, and DASI 450.

–code Gives the name of one of a set of "canned" tabs. The legal codes and their meanings are as follows:

–a 1,10,16,36,72
Assembler, IBM S/370, first format

–a2 1,10,16,40,72
Assembler, IBM S/370, second format

–c 1,8,12,16,20,55
COBOL, normal format

–c2 1,6,10,14,49
COBOL compact format (columns 1-6 omitted). Using this code, the first typed character corresponds to card column 7, one space gets you to column 8, and a tab reaches column 12. Files using this tab setup should include a format specification as follows:

<:t –c2 m6 s66 d:>

–c3 1,6,10,14,18,22,26,30,34,38,42,46,50,54,58,62,67
COBOL compact format (columns 1-6 omitted), with more tabs than **–c2.** This is the recommended format for COBOL. The appropriate format specification is:

<:t –c3 m6 s66 d:>

–f 1,7,11,15,19,23
FORTRAN

–p 1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61
PL/I

–s 1,10,55
SNOBOL

–u 1,12,20,44
UNIVAC 1100 Assembler

In addition to these "canned" formats, three other types exist:

–n A repetitive specification requests tabs at columns 1+*n*, 1+2**n*, etc. Note that such a setting leaves a left margin of *n* columns on TermiNet terminals *only*. Of particular importance is the value **–8**: this represents the UNIX system "standard" tab setting, and is the most likely tab setting to be found at a terminal. Another special case is

the value **−0**, implying no tabs at all.

n1,n2,... The arbitrary format permits the user to type any chosen set of numbers, separated by commas, in ascending order. Up to 40 numbers are allowed. If any number (except the first one) is preceded by a plus sign, it is taken as an increment to be added to the previous value. Thus, the tab lists 1,10,20,30 and 1,10,+10,+10 are considered identical.

− −*file* If the name of a file is given, *tabs* reads the first line of the file, searching for a format specification. If it finds one there, it sets the tab stops according to it, otherwise it sets them as **−8**. This type of specification may be used to make sure that a tabbed file is printed with correct tab settings, and would be used with the *pr*(1) command:

tabs −− file; pr file

Any of the following may be used also; if a given flag occurs more than once, the last value given takes effect:

−T*type* *Tabs* usually needs to know the type of terminal in order to set tabs and always needs to know the type to set margins. *Type* is a name listed in *term*(5). If no **−T** flag is supplied, *tabs* searches for the **$TERM** value in the *environment* (see *environ*(5)). If no *type* can be found, *tabs* tries a sequence that will work for many terminals.

+m*n* The margin argument may be used for some terminals. It causes all tabs to be moved over *n* columns by making column *n+1* the left margin. If **+m** is given without a value of *n*, the value assumed is 10. For a TermiNet, the first value in the tab list should be 1, or the margin will move even further to the right. The normal (leftmost) margin on most terminals is obtained by **+m0**. The margin for most terminals is reset only when the **+m** flag is given explicitly.

Tab and margin setting is performed via the standard output.

DIAGNOSTICS

illegal tabs	when arbitrary tabs are ordered incorrectly.
illegal increment	when a zero or missing increment is found in an arbitrary specification.
unknown tab code	when a "canned" code cannot be found.
can't open	if **− −***file* option used, and file can't be opened.
file indirection	if **− −***file* option used and the specification in that file points to yet another file. Indirection of this form is not permitted.

SEE ALSO

pr(1).

environ(5), term(5) in the *3B2 Computer System Programmer Reference Manual.*

BUGS

There is no consistency among different terminals regarding ways of clearing tabs and setting the left margin.

It is generally impossible to usefully change the left margin without also setting tabs.

Tabs clears only 20 tabs (on terminals requiring a long sequence), but is willing to set 64.

NAME

time — time a command

SYNOPSIS

time command

DESCRIPTION

The *command* is executed; after it is complete, *time* prints the elapsed time during the command, the time spent in the system, and the time spent in execution of the command. Times are reported in seconds.

The times are printed on standard error.

SEE ALSO

times(2) in the *3B2 Computer System Programmer Reference Manual.*

NAME

tty – get the name of the terminal

SYNOPSIS

tty [**–l**] [**–s**]

DESCRIPTION

Tty prints the path name of the user's terminal.

–l prints the synchronous line number to which the user's terminal is connected, if it is on an active synchronous line.

–s inhibits printing of the terminal path name, allowing one to test just the exit code.

EXIT CODES

2 if invalid options were specified,
0 if standard input is a terminal,
1 otherwise.

DIAGNOSTICS

"not on an active synchronous line" if the standard input is not a synchronous terminal and **–l** is specified.
"not a tty" if the standard input is not a terminal and **–s** is not specified.

NAME

units — conversion program

SYNOPSIS

units

DESCRIPTION

Units converts quantities expressed in various standard scales to their equivalents in other scales. It works interactively in this fashion:

```
You have: inch
You want: cm
        * 2.540000e+00
        / 3.937008e-01
```

A quantity is specified as a multiplicative combination of units optionally preceded by a numeric multiplier. Powers are indicated by suffixed positive integers, division by the usual sign:

```
You have: 15 lbs force/in2
You want: atm
        * 1.020689e+00
        / 9.797299e-01
```

Units only does multiplicative scale changes; thus it can convert Kelvin to Rankine, but not Celsius to Fahrenheit. Most familiar units, abbreviations, and metric prefixes are recognized, together with a generous leavening of exotica and a few constants of nature including:

pi	ratio of circumference to diameter,
c	speed of light,
e	charge on an electron,
g	acceleration of gravity,
force	same as **g**,
mole	Avogadro's number,
water	pressure head per unit height of water,
au	astronomical unit.

Pound is not recognized as a unit of mass; **lb** is. Compound names are run together, (e.g., **lightyear**). British units that differ from their U.S. counterparts are prefixed thus: **brgallon**. For a complete list of units, type:

```
cat /usr/lib/unittab
```

FILES

/usr/lib/unittab

NAME

xargs – construct argument list(s) and execute command

SYNOPSIS

xargs [flags] [command [initial-arguments]]

DESCRIPTION

Xargs combines the fixed *initial-arguments* with arguments read from standard input to execute the specified *command* one or more times. The number of arguments read for each *command* invocation and the manner in which they are combined are determined by the flags specified.

Command, which may be a shell file, is searched for, using one's **$PATH**. If *command* is omitted, **/bin/echo** is used.

Arguments read in from standard input are defined to be contiguous strings of characters delimited by one or more blanks, tabs, or new-lines; empty lines are always discarded. Blanks and tabs may be embedded as part of an argument if escaped or quoted. Characters enclosed in quotes (single or double) are taken literally, and the delimiting quotes are removed. Outside of quoted strings a backslash (\) will escape the next character.

Each argument list is constructed starting with the *initial-arguments*, followed by some number of arguments read from standard input (Exception: see **–i** flag). Flags **–i**, **–l**, and **–n** determine how arguments are selected for each command invocation. When none of these flags are coded, the *initial-arguments* are followed by arguments read continuously from standard input until an internal buffer is full, and then *command* is executed with the accumulated args. This process is repeated until there are no more args. When there are flag conflicts (e.g., **–l** vs. **–n**), the last flag has precedence. *Flag* values are:

–l*number*
: *Command* is executed for each non-empty *number* lines of arguments from standard input. The last invocation of *command* will be with fewer lines of arguments if fewer than *number* remain. A line is considered to end with the first new-line *unless* the last character of the line is a blank or a tab; a trailing blank/tab signals continuation through the next non-empty line. If *number* is omitted, 1 is assumed. Option **–x** is forced.

–i*replstr*
: Insert mode: *command* is executed for each line from standard input, taking the entire line as a single arg, inserting it in *initial-arguments* for each occurrence of *replstr*. A maximum of 5 arguments in *initial-arguments* may each contain one or more instances of *replstr*. Blanks and tabs at the beginning of each line are thrown away. Constructed arguments may not grow larger than 255 characters, and option **–x** is also forced. {} is assumed for *replstr* if not specified.

–n*number*
: Execute *command* using as many standard input arguments as possible, up to *number* arguments maximum. Fewer arguments will be used if their total size is greater than *size* characters, and for the last invocation if there are fewer than *number* arguments remaining. If option **–x** is also coded, each *number* arguments must fit in the *size* limitation, else *xargs* terminates execution.

−t Trace mode: The *command* and each constructed argument list are echoed to file descriptor 2 just prior to their execution.

−p Prompt mode: The user is asked whether to execute *command* each invocation. Trace mode (**−t**) is turned on to print the command instance to be executed, followed by a **?...** prompt. A reply of **y** (optionally followed by anything) will execute the command; anything else, including just a carriage return, skips that particular invocation of *command*.

−x Causes *xargs* to terminate if any argument list would be greater than *size* characters; **−x** is forced by the options **−i** and **−l**. When neither of the options **−i**, **−l**, or **−n** are coded, the total length of all arguments must be within the *size* limit.

−s*size* The maximum total size of each argument list is set to *size* characters; *size* must be a positive integer less than or equal to 470. If **−s** is not coded, 470 is taken as the default. Note that the character count for *size* includes one extra character for each argument and the count of characters in the command name.

−e*eofstr* *Eofstr* is taken as the logical end-of-file string. Underbar (_) is assumed for the logical **EOF** string if **−e** is not coded. The value **−e** with no *eofstr* coded turns off the logical **EOF** string capability (underbar is taken literally). *Xargs* reads standard input until either end-of-file or the logical **EOF** string is encountered.

Xargs will terminate if either it receives a return code of **−1** from, or if it cannot execute, *command*. When *command* is a shell program, it should explicitly *exit* (see *sh*(1)) with an appropriate value to avoid accidentally returning with **−1**.

EXAMPLES

The following will move all files from directory $1 to directory $2, and echo each move command just before doing it:

```
ls $1 | xargs −i −t mv $1/{} $2/{}
```

The following will combine the output of the parenthesized commands onto one line, which is then echoed to the end of file *log*:

```
(logname; date; echo $0 $*) | xargs >>log
```

The user is asked which files in the current directory are to be archived and archives them into *arch* (1.) one at a time, or (2.) many at a time.

```
1. ls | xargs −p −l ar r arch
2. ls | xargs −p −l | xargs ar r arch
```

The following will execute *diff*(1) with successive pairs of arguments originally typed as shell arguments:

```
echo $* | xargs −n2 diff
```

SEE ALSO

sh(1).

Index

A

B

C

D

E

F

H

L

M

N

P

R

S

T

U

V

X