XELOS USER GUIDE

48-165 F00 R01

The information in this document is subject to change without notice and should not be construed as a commitment by The Concurrent Computer Corporation. The Concurrent Computer Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and it may be used or copied only in a manner permitted by that license. Any copy of the described software must include any copyright notice, trademarks, or other legends or credits of The Concurrent Computer Corporation and/or its suppliers. Title to and ownership of the described software and any copies thereof shall remain in The Concurrent Computer Corporation and/or its suppliers.

The licensed programs described herein may contain certain encryptions or other devices which may prevent or detect unauthorized use of the Licensed Software. Temporary use permitted by the terms of the License Agreement may require assistance from The Concurrent Computer Corporation.

The Concurrent Computer Corporation assumes no responsibility for the use or reliability of the software on equipment that is not supplied by Concurrent Computer Corporation.

XELOS™ is a Concurrent Computer Corporation licensed product derived from UNIX® System V Release 2.0 and DOCUMENTER'S WORKBENCH™ under license from AT&T.

XELOS is a trademark of The Concurrent Computer Corporation

UNIX® is a registered trademark of AT&T Bell Laboratories

DOCUMENTER'S WORKBENCH is a trademark of AT&T Technologies

IMAGEN is a trademark of the IMAGEN Corporation

© 1984 AT&T Technologies - All Rights Reserved

© 1987 The Concurrent Computer Corporation - All Rights Reserved

The Concurrent Computer Corporation
2 Crescent Place
Oceanport, New Jersey 07757

Printed in the United States of America

This manual was typeset on an IMAGEN™IMPRINT 8/300 laser
printer driven by the *troff* formatter operating

CONTENTS

Chapter 1 INTRODUCTION

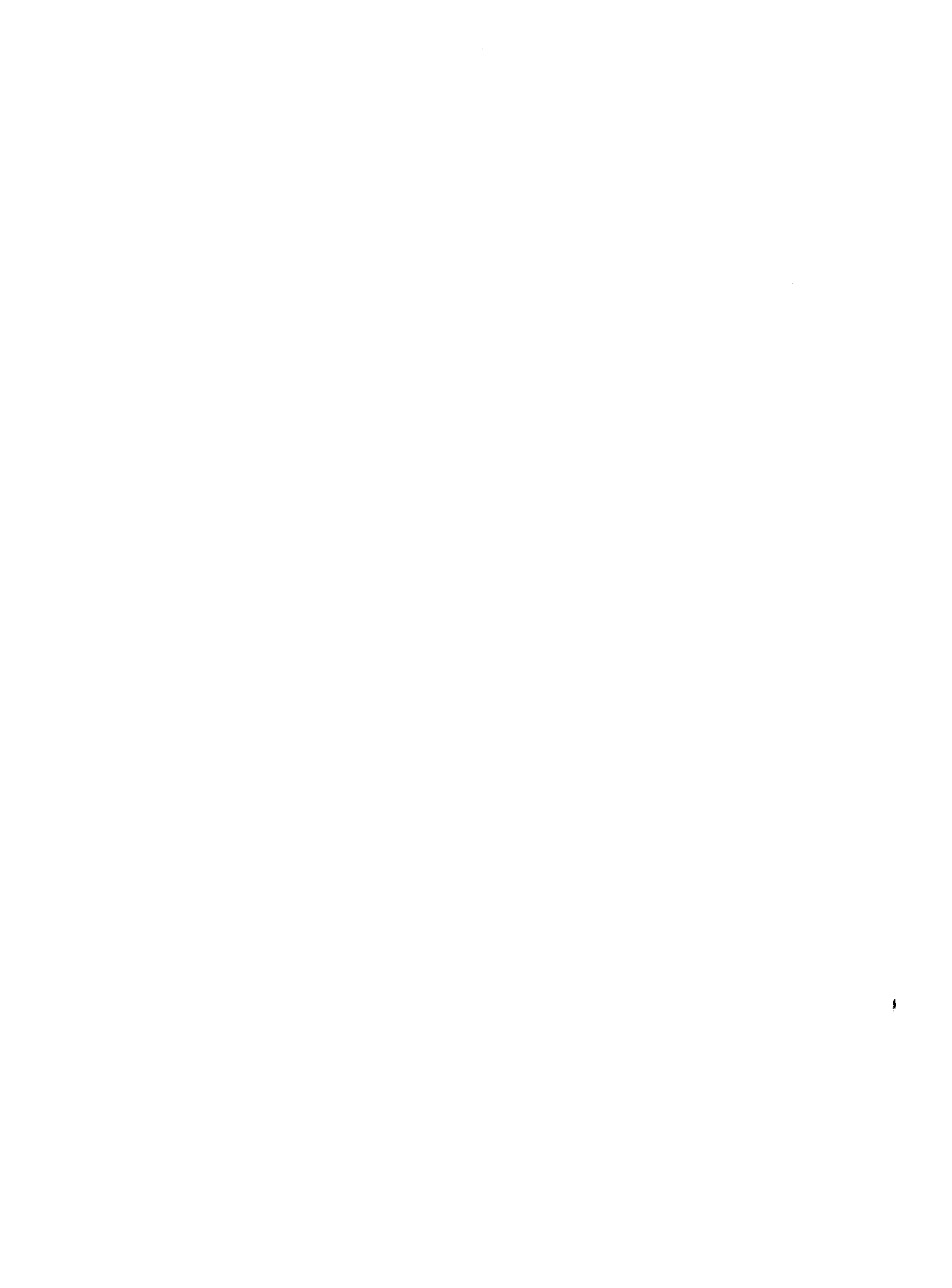
Chapter 2 PRIMER

Chapter 3 BASICS FOR BEGINNERS

Chapter 4 TUTORIAL—TEXT EDITOR

Chapter 5 AN INTRODUCTION TO THE SHELL

Chapter 6 GLOSSARY



Chapter 1

INTRODUCTION

PAGE

GENERAL 1

CAVEATS 2

Chapter 1

INTRODUCTION

GENERAL

The *XELOS* User Guide* covers the following topics:

- A description of the features in the XELOS operating system
- A general overview of the capabilities of the XELOS operating system
- Instructions on how to use the XELOS operating system.

Not all of the capabilities of the XELOS operating system are described or illustrated herein, but enough are described so that a new user can become familiar with the use of the XELOS operating system.

Using the available information, users who have more interest than the novice can utilize the information herein to accomplish their tasks with some experimenting and self-teaching.

Throughout this volume, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *XELOS Administrator Reference Manual*. Other references to entries of the form **name**(N), where "N" is a number (1 or 6) possibly followed by a letter, refer to entry **name** in section N of the *XELOS User Reference Manual*. Entries where "N" is a number (2 through 5) possibly followed by a letter, refer to entry **name** in section N of the *XELOS Programmer Reference Manual*.

---

* XELOS is a trademark of The Perkin-Elmer Corporation

CAVEATS

Document processing features described throughout the following sections may not be available on your system. Contact your system administrator to see if the DOCUMENTER'S WORKBENCH* software option has been installed.

---

* DOCUMENTER'S WORKBENCH is a trademark of AT&T Technologies, Inc.

Chapter 2

PRIMER

PAGE

INTRODUCTION 1

HUMAN INTERFACE 2

Concept of a Login 2

Logging In 3

Logging Off 5

Entering Commands 5

Stopping a Program 10

Mail 10

Writing to Other Users 11

On-line Manual 13

Chapter 2

PRIMER

INTRODUCTION

This section of the *XELOS User Guide* provides the information that users will need to access the XELOS operating system. It is not intended to be a detailed description. Many of the subjects described are discussed in detail in other sections of this volume or the *XELOS User Reference Manual.*

Throughout this section, each reference of the form **name(1M)**, **name(7)**, or **name(8)** refers to entries in the *XELOS Administrator Reference Manual.* Other references to entries of the form **name(N)**, where "N" is a number (1 or 6) possibly followed by a letter, refer to entry **name** in section N of the *XELOS User Reference Manual.* Entries where "N" is a number (2 through 5) possibly followed by a letter, refer to entry **name** in section N of the *XELOS Programmer Reference Manual.*

In this primer, software programs that can be executed by users are referred to as **programs**. A program that is in some state of execution is referred to as a **process**. The request typed by the user is referred to as a **command** or "**command line.**"

In this primer, the following graphic conventions are used in examples:

(RETURN) — Indicates that the user should press the RETURN key on the terminal keyboard.

(DEL) — Indicates that the user should press the key marked DEL, DELETE, or RUBOUT (whichever is appropriate for the terminal being used).

HUMAN INTERFACE

Concept of a Login

The XELOS operating system is accessed by the use of a **login**. A login is used by the system to uniquely identify users. Before the user can access the system, the user must be assigned a login by the system administrator. Every login consists of the following components:

login name

user identification number (uid)

group identification number (gid)

password.

A **login name** is a unique string of letters (should be all lower-case) and/or numbers that identifies an individual to the system. The login name must begin with a letter. In many cases, a person's login name is their real first name, last name, initials, or nickname. Any string of letters and/or digits can be used as your login name, as long as it is **unique** (i.e., different from all other login names). Only the first eight characters of a login name are used by the system. Login names are assigned by the system administrator.

The **uid** of a login is a unique number assigned to each login by the system administrator. This number is used by the system to identify the owners of information stored on the system and the commands that users are executing.

The **gid** is a unique number assigned by the system administrator to each group. This number identifies **groups** of users that have something in common. For example, all logins used by people in the same department (or working on the same project) may have the same gid. The gid is important for security and accounting reasons. The impact of gid numbers

on the user and the group that the user belongs to is described later.

The **password** is a string of 13 characters chosen from a 64-character alphabet (., \, 0-9, A-Z, a-z) that serves to control access to a login. The password for a login is the main security feature of the XELOS operating system. Usually, every login is assigned a password. When a user **logs in** to the system, the password (if any) assigned to the login being used is requested. Access to the system is not permitted until the correct password is entered. The user can change a password as needed to ensure that others are not accessing the user's login (and consequently the user's data). Any string of letters, numbers, etc., can be used as a password as long as it is from six to thirteen characters in length and composed of upper-case letters, lower-case letters, numbers, or punctuation.

It is recommended that obvious strings such as the user's social security number, birth date, or other data that could be well known about the user not be used as passwords. If the password is something that is well known about the user, someone could gain access to the user's login with little effort. The more unusual your password, the more effective your security.

Logging In

In order to log in, the power to the terminal must be turned on and the appropriate switches set. Depending on the type of terminal and communication link, the user may need to press the return or break key a couple of times. This is to synchronize your terminal with the system. When communication is established, the system will prompt with:

```
login:
```

The user should type in his/her login name followed by a return. After the system digests your login name, it will prompt for your password with:

Password:

The user should then type his/her password followed by a return. The system does not echo your password on the terminal as you type it in. This is an extra security measure. If you entered your login name and password correctly, the system may print one or more "messages of the day". Following the messages, the system will prompt you with the primary prompt string, which is usually the *$* symbol. If a mistake is made while logging in or the system administrator has not set up the user's login on the system, the following error message is printed:

login incorrect

This error message is followed by the login: message. The user should attempt to log in again.

The XELOS operating system has a hierarchy of *directories*. When the system administrator gave the user a login name, the administrator also created a "directory" for the user. This directory ordinarily is the same name as the user login name and is known as the *login* or *home* directory of the user. When the user logs in, the *home* directory becomes the current directory or working directory of the user. Any file created under the login name (assuming no other subdirectories have been created yet) is by default in the home directory. The user may, however, create one or more directories under the *home* directory. The user may then change to subdirectories by appropriate use of a "change directory" command. See **cd**(1) for details. Under a directory or a subdirectory, the user may create files as necessary. The user is the owner of the *home* directory and all subdirectories created under the *home* directory. As the owner, the user has full permission to create, alter, and remove (destroy) all files and subdirectories of the *home* directory. To change from one directory to another, the command **cd** is used.

Logging Off

After completing your work, it is best to log off the system. Before logging off, you should have received the prompt string "$ " from the system. That is, all your commands have been completed; and the system is ready for another command.

The preferred method for logging off is accomplished by typing an American Standard Code for Information Interchange (ASCII) end of text (EOT) character which is sometimes called the end of file (EOF). On most terminals, the EOT character is generated by holding down the "CONTROL" key and pressing the lower-case "d" key once. This is also referred to as a **CONTROL-d**. Regardless of the terminal type, the power to it should be turned off when the terminal is no longer needed. For a terminal connected via a phone line, you should hang up the phone.

Entering Commands

The XELOS operating system **shell** (command interpreter) serves as the interface between the user and the system. The **shell** accepts requests from the user in the form of a **command line** and invokes the appropriate program to fulfill the request. The **shell** prompts (i.e., notifies) the user when it is ready to accept another request. The prompt of the XELOS operating system **shell** is the primary prompt string which is by default "**$** " (a dollar sign followed by a space).

Command Line Syntax

Commands or requests to the **shell** are usually in the form of a single line, that is, a string of one or more words followed by a return. This single line request entered following the prompt is referred to as a "command line". The command line is divided into two major parts—the program

name and arguments.

The first word of the command line is the name of the program to be executed. This is referred to as the **command**. All subsequent words are *arguments* to the command. Arguments are used to provide information required by the program.

Spaces and tabs serve as the delimiters for words on the command line. That is, all characters on the command line up to the first space or tab are interpreted as the command. All characters between the first space (or tab) and the second space (or tab) is the first argument, etc. Thus, the syntax for the command line is:

```
command argument argument argument ... ...(RETURN)
```

When spaces or tabs are needed within a single argument, that argument is enclosed by **double** quote marks. For example, to execute a program that requires two arguments such as *john l* and *doe*. The first argument should be *john* and the initial *l*, that is, "john l". The second argument should be *doe*. The required command line in this case would be:

```
command "john l" doe(RETURN)
```

Correction and Deletion

All users are likely to make mistakes, especially when typing. The XELOS operating system provides two features to correct command lines. These features are *only* effective for the current line (i.e., you have not ended the line with a return yet).

The first correction feature is the erase character (by default, #), and the second correction feature is the kill character (by default, @). The erase character erases the character preceding it. For example, a command line entered as

caf#t the fik#le (RETURN)

actually is "cat the file". The first # erases the first f and the second # erases the k. The erase character can be used to erase a series of characters such as in

this####the cat had kittens (RETURN)

which results in "the cat had kittens". The entire word "this" is erased by the series of # characters following it. The first # erases the s, the second # erases the i, the third # erases the h, and the fourth # erases the t. If you miscount the number of erase characters you need, as in

this ###the cat had kittens (RETURN)

the result would be "ththe cat had kittens". The three erase characters erase the space, the s, and the i preceding them.

If the user needs to enter a # in the command line for some reason, preceding the # with the backslash character (\) will turn off the "erase last character" meaning of the #. For example, a command line entered as

thsi##is is the \#7# 7 cat (RETURN)

is actually "this is the # 7 cat".

The second correction feature is the kill character. The kill character deletes the entire current line. For example, the user enters the command line

command#####omma#####mmad argm##gmu##ment

when the user was trying to enter "command argument". This command line is so full of mistakes and corrections it is hard to determine if it is right. It would be best to delete the entire line and start over. The user can delete the line by ending it with an @ instead of a return. For example in this sequence

```
kat###catteh##he file######## the flie##e@
cat the file (RETURN)
```

the first line is deleted by the @ character. It is much easier to delete it and reenter it (as in the second line of the example).

If the @ character is needed in a line, the backslash character (\) should precede it. For example, entering the line

```
The kill character is a \@ (RETURN)
```

results in "The kill character is a @".

Strange Terminal Behavior

Sometimes you can get into a state where your terminal acts strangely. For example, each letter may be typed twice (terminal may be in the half-duplex mode) or the RETURN may not cause a line feed or a return to the left margin. The user can often change this by logging out and logging back in. If logging back in fails to correct the problem, check the following areas:

keyboard	Keys such as caps lock, local, block, etc. should not be depressed.

dataphone	For terminals connected via phone lines, the baud rate could be incorrect.
switches	The rear panel of your terminal normally has several switches used to control terminal operations. These switches should be set to be compatible with the XELOS operating system.

If all else fails, the description of the **stty**(1) command can be read to determine the appropriate action to take. To get intelligent treatment of tab characters (which are much used in the XELOS operating system) if your terminal does not have tabs, type the command

```
stty -tabs
```

and the system will convert each tab into sufficient blanks to space to the next 8-character field. If your terminal does have hardware tabs, the command **tabs**(1) will set the stops correctly for you.

Read—ahead

The XELOS operating system has full read-ahead, which means that the user can type as fast as desired, whenever the user wants, even when some command is already outputting on the terminal. If typing is done during output, the input characters will appear intermixed with the output characters, but they will be stored away and interpreted in the correct order. So the user can type several commands one after another without waiting for the first to finish or even begin.

Stopping a Program

Most programs can be stopped by typing the character "DEL" (perhaps called "delete" or "rubout" on your terminal). The "interrupt" or "break" key found on most terminals can also be used. In a few programs, like the text editor, "DEL" stops whatever the program is doing but leaves you in that program. Hanging up the phone with the talk button depressed will also stop most programs.

Mail

After logging in, the user may sometimes get the following message:

```
You have mail.
```

The XELOS operating system provides a postal system so you can communicate with other users of the system. To read your mail, type the following command:

```
mail
```

Your mail will be printed, one message at a time, most recent message first. After each message, **mail**(1) waits for you to say what to do with it. The two basic responses are **d**, which deletes the message, and RETURN, which does not (so it will still be there the next time you read your mailbox). Other responses are described in the *XELOS User Reference Manual*.

How is mail sent to someone else? Assume that "jones" is someone's login name which is recognized by **login**(1). The easiest way to send mail to "jones" is as follows:

mail jones
now type in the text of the letter
on as many lines as you like...
After the last line of the letter
type the character "CONTROL-d",
that is, hold down "CONTROL" and type
a letter "d".

The "CONTROL-d" sequence, often called end of file (EOF), is used throughout the system to mark the end of input from a terminal.

For practice, send mail to yourself. (This is not as strange as it might sound—mail to oneself is a handy reminder mechanism.)

There are other ways to send mail—you can send a previously prepared letter, and you can mail to a number of people all at once. For more details, see **mail(1)**.

Writing to Other Users

At some point, out of the blue will come a message like

Message from jones tty07...

which is accompanied by a startling beep on terminals that have the capability to beep. It means that Jones (jones) wants to talk to you, but unless you take explicit action, you will not be able to talk back. To respond, type the following command:

write jones

This establishes a two-way communication path. Now whatever jones

types on his terminal will appear on yours and vice versa. However, if you are in the middle of some program, you must get back to a state where you are talking to the command interpreter. Normally, whatever program you are running has to terminate or be terminated. If you are editing, you can escape temporarily from the editor—see the "Tutorial—Text Editor" section of this document. If you are printing and do not want this message in your printout or you simply do not want to be disturbed, enter the following:

```
mesg n
```

If you never wish to be disturbed, add the "**mesg n**" command line to your *.profile*.

A protocol is needed to keep what you type from getting garbled up with what jones types. Typically, a sequence like the following is used:

Jones types "write smith" and waits.

Smith types "write jones" and waits.

Jones now types a message
(as many lines as necessary).
When he is ready for a reply, he
signals it by typing
(o)
which stands for "over".

Now Smith types a reply, also
terminated by
(o).

This cycle repeats until
someone gets tired; he then
signals his intent to quit with
(oo)
for "over and out".

To terminate
the conversation, each side must
type a "CONTROL-d" character alone
at the beginning of a line. ("DELETE" also works.)
When the other person types "CONTROL-d",
you will get the message
EOF
on your terminal.

If you write to someone who is not logged in or who does not want to be disturbed, you will be told. If the target is logged in but does not answer after a decent interval, simply type "CONTROL-d".

On-line Manual

The *XELOS User Reference Manual*, *XELOS Programmer Reference Manual*, and *XELOS Administrator Reference Manual* are kept on-line. If you get stuck on something and cannot find an expert to assist you, you can print on your terminal some manual section that might help. This is also useful for getting the most up-to-date information on a command. To print a manual section, type "man command-name". Thus, to read up on the **who**(1) command, type

```
man who
```

and, of course,

```
man man
```

tells all about the **man**(1) command.

Chapter 3

BASICS FOR BEGINNERS

PAGE

DAY-TO-DAY USE 1

Creating Files—The Editor 1

What Files Are Out There? 2

Printing Files 4

Moving Files Around 6

What's in a File Name 7

What's in a File Name, Continued 11

Using Files for I/O Instead of the Terminal 16

Pipes 18

The Shell 20

DOCUMENT PREPARATION 22

Formatting Packages 23

Supporting Tools 25

Hints for Preparing Documents 27

Programming 28

Shell Programming 28

Programming with Shell 30

Programming in C 31

Other Languages 32

Chapter 3

BASICS FOR BEGINNERS

DAY-TO-DAY USE

Creating Files—The Editor

If you have to type a paper, a letter, or a program, how do you get the information into the machine? These tasks can be performed using the XELOS operating system "text editor". See **ed(1)** and the "TUTORIAL—TEXT EDITOR" section of this volume for a detailed description.

Throughout this section, each reference of the form **name**(N) refers to entries in the *XELOS Administrator Reference Manual*, *XELOS User Reference Manual*, or *XELOS Programmer Reference Manual.*

The XELOS operating system "text editor" operates on a "file". Simply stated, a file is just a collection of information stored in the machine. The following text will describe how to make some *files*. To create a file called *junk* with text in it, do the following:

ed junk (invokes the text editor)
?junk (indicates a new file named junk)
a (command to "**ed**" to add text)
now type in
whatever text you want ...
. (signals the end of text addition)

The "." that signals the end of adding text must be at the beginning of a

line by itself. Do not forget it, for until it is typed, no other **ed** commands will be recognized—everything you type will be treated as text to be added. Also note that no system prompt appears while you are appending, inserting, or changing text while in the text editor.

After a file exists, the user can do various editing operations on the text which was typed in, such as correcting spelling mistakes, rearranging paragraphs, etc. Finally, the user must write the information typed into a file with the editor command:

```
w
```

The **ed** will respond with the number of characters it wrote into the file *junk*.

Nothing is stored permanently in the *junk* file until the **w** command is used. If the user is editing a file and terminates before using the **w** command, the changes are not stored in the working file. The data in this case is saved in a file called *ed.hup* which the user can continue working with at the next editing session. But after **w**, the information is there permanently. The user can reaccess it any time by typing the following:

```
ed junk
```

Type a **q** command to quit the editor. (If you try to quit without writing, the text editor will print a "?" to remind you. A second **q** gets the user out of the text editor regardless.) Now create a second file called *temp* in the same manner. You should now have two files, *junk* and *temp*.

What Files Are Out There?

The **ls**(1) command lists the names (not contents) of any of the files that the XELOS operating system knows about. If you type

```
ls
```

the response will be

```
junk
temp
```

which are, indeed, the two files just created.

The names are sorted into alphabetical order automatically, but other variations are possible. For example, the command

```
ls −t
```

causes the files to be listed in the order in which they were last changed, most recent first. The −l option gives a "long" listing and is used as follows

```
ls −l
```

to produce something like

```
-rw-rw-rw- 1 bwk  bsk 41 Jul 22 02:56 junk
-rw-rw-rw- 1 bwk  bsk 78 Jul 22 12:57 temp
```

The date and time is the date and time of the last change to the file. The 41 and 78 are the number of characters (which should agree with the numbers you got from **ed**). The "bwk" is the owner of the file; i.e., the person who created it. The "bsk" identifies the group associated with "bwk". The "-rw−rw−rw−" determines who has permission to read, write, or execute the file. In this case, the owner, group, and others all have permission to read (r) and write (w). Note that there is no permission for anyone to execute (x). The first character in "-

rw−rw−rw−" is a "-" which indicates this is a file of data. A "d" in the first character would indicate a directory. The remaining nine characters are divided into three sets of permissions. Each set consists of three characters. The three sets correspond to the permissions of the owner, group, and all other users.

Options can be combined: **ls −lt** gives the same thing as **ls −l** but sorted into time order. The user can also list the files by name, and **ls** will list the information about them only. More details can be found in **ls**(1).

The use of optional arguments that begin with a minus sign (like *−t* and *−lt*) is a common convention for XELOS system programs. In general, if a program accepts such optional arguments, they precede any file name arguments. It is also vital that you separate the various arguments with spaces: **ls−l** is not the same as **ls −l** since the command **ls** must be separated from its argument *−l* by a space.

Printing Files

Now that you've created a file of text, how can the file be printed so people can look at it? There are several ways to print a file. One simple way to obtain a print is to use the editor, since printing is often done just before making changes anyway. The editor is used to print as follows:

```
ed junk
1,$p
```

The **ed** will reply with the count of the characters in *junk* and then print all the lines in the file. The user can also be selective about the parts of a file to be printed as follows:

```
ed junk
20,35p
```

There are times when it's not feasible to use the editor for printing. For example, there is a limit on how big a file **ed** can handle (several thousand lines). Secondly, it will only print one file at a time; and sometimes you want to print several, one after the other. So here are a couple of alternatives.

The simplest of all the printing programs is **cat(1)**. The **cat** command simply prints on the terminal the contents of all the files named and in the order listed. Thus the files are concatenated and printed. For example:

```
cat junk
```

prints one file, and

```
cat junk temp
```

prints two files. The files are simply concatenated onto the terminal.

The **pr**(1) command produces formatted printouts of files. As with **cat**, **pr** prints all the files named in a list. The difference is that it produces headings with date, time, page number, and file name at the top of each page, and extra lines to skip over the fold in the paper.

Thus,

```
pr junk temp
```

will print *junk* neatly, then skip to the top of a new page and print *temp* neatly.

The **pr** command can also produce multicolumn output. Inputing

```
pr -3 junk
```

prints *junk* in 3-column format. You can use any reasonable number in place of "3", and **pr** will do its best. The **pr** command has other capabilities also. See **pr**(1) for more information.

It should be noted that **pr** is *not* a formatting program in the sense of shuffling lines around and justifying margins. The true formatters are **nroff** and **troff**, which we will get to in the section on document preparation.

There are also programs that print files on a hard copy printer. See **lp**(1) for more information.

Moving Files Around

The user is ready for bigger things after gaining experience in creating and printing files. For example, the user can move a file from one place to another (which amounts to giving it a new file name), like this:

```
mv junk precious
```

This means that what used to be named *junk* is now named *precious*. An **ls**(1) command would now result in the following:

```
precious
temp
```

The contents of *junk* are now in *precious*. Notice that the *junk* file no longer exists. Beware that if you move a file to another one that already exists, the already existing file contents are lost *forever*.

If you want to make a copy of a file (i.e., to have two versions of something), use the **cp(1)** command as follows:

```
cp precious temp1
```

This makes a duplicate copy of *precious* in *temp1.*

When you are finished creating and moving files, the files can be removed from the file system by the **rm(1)** command. The command is used as follows:

```
rm temp temp1
```

This will remove both the *temp* and *temp1* files.

The user will get a warning message if one of the named files is not there, but **rm**, like most XELOS system commands, does its work silently. There is no prompting or response, and error messages are just occasionally shortened. This terseness is sometimes disconcerting to newcomers, but experienced users find it desirable.

What's in a File Name

So far we have used file names without ever saying what is a legal name, so it is time for a couple of rules. First, file names are limited to 14 characters, which is enough to be descriptive. Second, although any character can be used in a file name, common sense dictates sticking to ones that are visible and avoiding characters that could be used with other meanings. We have already seen, for example, that in the **ls(1)** command, **ls −t** means to list in time order. So if a file existed whose name was *−t*, you would have a tough time listing it by name. Besides the minus sign, there are other characters which have special meaning. To avoid pitfalls, use only letters, numbers, and the period until you are familiar with the

system.

On to some more positive suggestions. Suppose you are typing a large document like a book. Logically, this divides into many small pieces, like chapters and perhaps sections. Physically, it must be divided too, for ed will not handle really big (over 90,000 characters) files. Thus, the document should be typed as a number of files. One possible method is to have a separate file for each chapter as follows:

```
chap1
chap2
etc. ...
```

Another method is breaking each chapter into several files as follows:

```
chap1.1
chap1.2
chap1.3
...
chap2.1
chap2.2
...
```

It can now be determined at a glance where a particular file fits into the whole.

There are advantages to a systematic naming convention which are not obvious to the novice XELOS system user. To print the whole book, the user could enter the following:

```
pr chap1.1 chap1.2 chap1.3 ...
```

Using the **pr**(1) command like this would be tiring and possibly lead to making mistakes. Fortunately, there is a shortcut. The user can enter:

```
pr chap*
```

The * means "anything at all", so this translates into "print all files whose names begin with *chap* listed in alphabetical order".

This shorthand notation is not a property of the **pr** command by the way. It is system-wide, a service of the program that interprets commands—the "**shell**", **sh**(1). The files in the book can be listed by using

```
ls chap*
```

which produces the following:

```
chap1.1
chap1.2
chap1.3
...
```

The * is not limited to the last position in a file name. The * can be used anywhere and can occur several times. Thus, entering

```
rm *junk* *temp*
```

removes all files that contain *junk* or *temp* as any part of their name. As a special case, * by itself matches every file name, so

```
pr *
```

prints all your files (alphabetical order), and

```
rm *
```

removes *all files*. (Before using the **rm** * command, make sure all files are not needed!)

The * is not the only pattern-matching feature available. To print only chapters 1 through 4 and 9, use the following command:

```
pr chap[12349]*
```

The **[...]** means to match any of the characters inside the brackets. A range of consecutive letters or digits can be abbreviated as follows:

```
pr chap[1-49]*
```

Letters can also be used within brackets. The **[a-z]** pattern-matching feature matches any character in the range *a* through *z*.

The **?** pattern matches any single character, so

```
ls ?
```

lists all files which have single-character names, and

```
ls -l chap?.1
```

lists information about the first file of each chapter *chap1.1*, *chap2.1*, etc.

Of these niceties, * is certainly the most useful to become familiar with. The others are frills, but worth knowing.

If the special meaning of *, ?, etc. needs to be turned off, enclose the entire argument in single quotes as follows:

```
ls '?'
```

Some examples of this will be shown in the following paragraphs.

What's in a File Name, Continued

When the file called *junk* is first created, how does the system know that there is not another *junk* somewhere else, especially since the person in the next office could also be reading this tutorial? The answer is that generally each user has a private *directory*, which contains only the files that belong to that particular user. When you login, you are "in" your directory. Unless the user takes special action when creating a new file, the new file is made in the directory that the user is currently in. This is most often your own directory, and thus the file is unrelated to any other file of the same name that might exist in another (someone else's) directory.

The set of all files is organized into a (usually big) tree with your files located several branches into the tree. It is possible for you to "walk" around this tree and find any file in the system by starting at the root of the tree and walking along the proper set of branches. Conversely, you can start at your present location and walk toward the root.

Try the latter first. The basic tool is the command **pwd**(1) (print working directory) which prints the name of the directory the user is currently in.

Although the details will vary according to the system the user is on, the **pwd**(1) command will print something like:

```
/usr/your_name
```

This message indicates that the user is currently in the directory *your_name*, which is in turn in the directory */usr*, which is in turn in the root directory called by convention just /. (Even if it is not called */usr* on your system, the message will be something analogous. Recognize any differences between your machine's pathname and the standard setup and

make the corresponding changes to the following command lines when appropriate.)

If user now types

```
ls /usr/your_name
```

the results should be exactly the same list of file names as obtained from a plain **ls**(1). With no arguments, **ls** lists the contents of the current directory. Given the name of a directory, it lists the contents of that directory.

Next, try using the following command:

```
ls /usr
```

This should print a long series of names, among which is your own login name *your_name*. On many systems, *usr* is a directory that contains the directories of all the normal users of the system.

The next step is to try the following:

```
ls /
```

The response should be something like this (although again the details may be different):

```
bin
dev
etc
lib
tmp
usr
```

This is a collection of the basic directories of files that the system knows about; we are at the root of the tree.

If *junk* is still in your directory, enter the following:

```
cat /usr/your_name/junk
```

The name

```
/usr/your_name/junk
```

is called the **pathname** of the file that is normally thought of as *junk*. The **pathname** represents the full name of the path as followed from the root through the tree of directories to get to a particular file. It is a universal rule in the XELOS operating system that anywhere an ordinary file name can be used, the **pathname** can also be used.

This is not too exciting if all the files of interest are in your own directory; but if you work with someone else or on several projects concurrently, it becomes handy indeed. For example, your friends can print your book by entering the following:

```
pr /usr/your_name/chap*
```

Similarly, you can find out what files your neighbor has by entering:

```
ls /usr/neighbor
```

The "neighbor" just entered represents the login name of your neighbor. A copy of one of your neighbor's files can be made as follows:

```
cp /usr/neighbor/his_file your_file
```

BASICS FOR BEGINNERS

If a file owner does not want someone else to have access to the owner's files or vice versa, privacy can be arranged. Each file and directory has read-write-execute (rwx) permissions for the owner, a group, and everyone else, which can be set to control access. See **ls**(1) and **chmod**(1) for details. Most users find openness of more benefit than privacy (most of the time).

As a final experiment with pathnames, try the following:

```
ls /bin /usr/bin
```

Do some of the names look familiar? When a program is run by typing its name after the prompt character, the system simply looks for a file of that name. It normally looks first in your directory (where it typically does not find it), then in */bin* and finally in */usr/bin*. There is nothing magic about commands like **cat**(1) or **ls**(1), except that they have been collected into a couple of places to be easy to find and administer.

It is possible for two or more users to work regularly with common information in the same document. This common document should be divided up into several files. To prevent users from working in the same file at the same time, the users should be allowed to work only on specified files. The files that make up this common document can be located in the directories of several users. These files can be combined into one document using the copy command [**cp**(1)] or the .so macro. If this common document is to be located in the same directory, the users can change the current working directory as follows:

```
cd full_path_name
```

Now you are ready to edit your specified files in this directory.

Another method of working on the same document is to locate the files in your friend's directory and login as your friend. Take into consideration that this defeats the accounting purpose of individual logins. If you are already logged in as yourself and want to work in a friend's files, change

the current working directory as follows:

```
cd /usr/your_friend
```

Now when a file name is used in something like **cat**(1) or **pr**(1), the command refers to the file in your friend's directory. Changing directories does not affect any permissions associated with a file. If you cannot access a file, get the owner to change permissions via **chmod**(1). Of course, if you forget what directory you are in, type

```
pwd
```

to find out.

It is usually convenient to arrange your own files so that all the files related to one thing are in a directory separate from other projects. For example, when writing your book, the user might want to keep all the text in a directory called *book*. A directory can be made using the **mkdir**(1) command. The *book* directory is made as follows:

```
mkdir book
```

The *book* directory can now be accessed to input chapters as follows:

```
cd book
```

If you logged in as yourself, the **pathname** of *book* is:

```
/usr/your_name/book
```

To remove the *book* directory, type:

```
rm book/*
rmdir book
  or
rm −r book
```

The **rm book/*** command removes all files in the *book* directory. The **rmdir book** command is then used to remove the empty directory. The *book* directory must be empty before the **rmdir** command will work. The **rm −r book** command recursively deletes the entire contents of the *book* directory and then removes the *book* directory itself.

The user can go up one level in the tree of files by entering:

```
cd ..
```

The ".." is the name of the parent of whatever directory you are currently in. For completeness, "." is an alternate name for the directory you are in.

Using Files for I/O Instead of the Terminal

Most of the commands used so far produce output on the terminal. Other commands, like the editor, take input from the terminal. It is universal in XELOS systems that the terminal can be replaced by a file for either or both of input and output.

As one example,

```
ls
```

makes a list of files on your terminal. But if the user enters

```
ls >filelist
```

a list of your files will be placed in the file *filelist* (which will be created if it does not already exist or overwritten if it does). The symbol > means "put the output on the following file rather than on the terminal". Nothing is produced on the terminal. As another example, the user could combine several files into one by capturing the output of **cat** in a file:

```
cat f1 f2 f3 >temp
```

Another symbol, that operates very much like > does, is >>. The >> means "add to the end of". That is,

```
cat f1 f2 f3 >>temp
```

means to concatenate *f1*, *f2*, and *f3* to the end of whatever is already in *temp* instead of overwriting the existing contents. As with >, if *temp* does not exist, it will be created.

In a similar way, the symbol < means to take the input for a program from the following file instead of from the terminal. Thus, the user could make up a script of commonly used editing commands and put them into a file called *script*. The script could then be run on a file by entering:

```
ed file <script
```

Another example is using **ed** to prepare a letter in file *let*. The letter (file *let*) could then be sent to several people as follows:

```
mail adam eve mary joe <let
```

Pipes

One of the novel contributions of the XELOS operating system is the idea of a **pipe**. A pipe is simply a way to connect the output of one program to the input of another program, so the two run as a sequence of processes—a pipeline.

For example,

```
pr f g h
```

will print the files *f*, *g*, and *h*, beginning each on a new page. Instead of printing the files separately, the files can be printed together as follows:

```
cat f g h >temp
pr <temp
rm temp
```

This method is more work than necessary. To take the output of **cat** and connect it to the input of **pr**, use the following pipe:

```
cat f g h | pr
```

The vertical bar | means to take the output from **cat** which would normally have gone to the terminal and put it into **pr** to be neatly formatted.

There are many other examples of pipes. For example,

```
ls | pr -3
```

prints a list of your files in three columns. The program **wc(1)** counts the number of lines, words, and characters in its input; while the **who(1)** command prints a list of users currently logged on the system, one per access port.

Thus, the command line

```
who | wc -l
```

tells how many people are logged on. And of course

```
ls | wc -l
```

counts your files.

Most programs that read from the terminal can read from a pipe instead. Most programs that write on the terminal can write on a pipe instead. There can be as many commands in a pipeline as desired.

Many XELOS operating system programs are written to take input from one or more files if file arguments are given. If no arguments are given, the programs will read from the terminal, and thus can be used in pipelines. One example using the **pr**(1) command to print files *a*, *b*, and *c* in three columns and in the order specified is as follows:

```
pr -3 a b c
```

But in

```
cat a b c | pr -3
```

the **pr** prints the information coming down the pipeline, still in three columns.

BASICS FOR BEGINNERS

The Shell

The mysterious "**shell**" mentioned previously is actually the **sh**(1) command. The shell is the program that interprets what is typed as commands and arguments. The shell also looks after translating *, etc., into lists of file names, and <, >, and | into changes of input and output streams.

The shell has other capabilities too. For example, the user can run two programs with one command line by separating the commands with a semicolon. The shell recognizes the semicolon and breaks the line into two commands. Thus

```
date; who
```

does both commands before returning with a prompt character.

More than one program can run *simultaneously* if desired. This is beneficial when doing something time-consuming, like using the editor script. The act of running programs in the background prevents waiting around for the results before starting something else. An example follows:

```
ed file <script &
```

The ampersand (&) at the end of a command line means "start this command running, then take further commands from the terminal immediately", that is, don't wait for it to complete. Thus the script will begin, but the user can do something else at the same time. Of course, to keep the output from interfering with what you are doing on the terminal, it would be better to enter

```
ed file <script >script.out &
```

which saves the output lines in a file called *script.out*.

When a command is initiated with **&**, the system replies with a number called the process number. Programs running simultaneously can be terminated as follows:

```
kill process_number
```

The process number is used to identify the command to be stopped. If you forget the process number, the **ps**(1) command will list the process number for all programs you are running. (Entering **kill 0** will kill all your processes.) If you are curious about other people, **ps −a** will provide information about all *active* programs that other users are running.

To start three commands that will execute in the order specified and in the background, enter the following:

```
(command_1; command_2; command_3) &
```

A background pipeline can be started as follows:

```
command_1 | command_2 &
```

Just as the editor or some similar program can get its input from a file instead of from the terminal, the shell can read a file to get commands. For example, suppose the user wants to perform a sequence of actions after every login such as:

- Set the tabs on the terminal
- Find out the date
- Find out who's on the system.

The three necessary commands to perform these actions [**tabs**(1), **date**(1),

and **who**(1)] could be put in a file called *startup*. The *startup* file would then be run as follows:

```
sh startup
```

This instruction commands the machine to run the shell with the file *startup* as input. The effect is the same as typing the contents of *startup* on the terminal.

If this is to be a regular thing, the need to type **sh** every time can be eliminated by typing the following command only once:

```
chmod +x startup
```

To run the sequence of commands thereafter, the user only needs to enter:

```
startup
```

The **chmod**(1) command marks the file as being executable. The shell recognizes this and runs it as a sequence of commands.

If the user wants *startup* to run automatically for every login, create a file in your login directory called *.profile* and place in it the line "startup". Upon logging in, the shell gains control and executes the commands found in the *.profile* file. We will get back to the shell in the section on programming.

DOCUMENT PREPARATION

XELOS operating systems are used extensively for document preparation. There are two major formatting programs, that is, programs that produce a text with justified right margins, automatic page numbering and titling,

automatic hyphenation, etc. The **nroff** program is designed to produce output on terminals and line-printers. The **troff** (pronounced "tee-roff") program was designed to drive a phototypesetter, which produces very high quality output on photographic paper. This document was formatted with **troff**. The document preparation packages may or may not be installed on your XELOS system. Check with your system administrator to see if your system contains the DOCUMENTER'S WORKBENCH software package.

Formatting Packages

The basic idea of **nroff** and **troff**(1) is that the text to be formatted contains within it "formatting commands" that indicate in detail how the formatted text is to look. For example, there may be commands that specify how long lines are, whether to use single or double spacing, and the running titles to use on each page.

Because **nroff** and **troff** are relatively hard to learn to use effectively, several "packages" of canned formatting requests are available to let you specify paragraphs, running titles, footnotes, multicolumn output, etc. with little effort and without having to learn **nroff** and **troff**. These packages take a modest effort to learn, but the rewards for using them are so great that it is time well spent.

This section provides a brief description of the "memorandum macros" package known as **mm**(1). Formatting requests typically consist of a period and two upper-case letters, such as

```
.TL
```

which is used to introduce a title or

```
.P
```

to begin a new paragraph.

The text of a typical document is entered so it looks something like this:

```
.TL
title
.AU "author information"
.MT "memorandum type"
.P
Enter text ---
---
.P
More text ---
---
.SG "signature"
```

The lines that begin with a period are the formatting macro requests. For example, **.P** calls for starting a new paragraph. The precise meaning of **.P** depends on the output device being used (typesetter or terminal, for instance) and the publication the document will appear in. For example, **−mm** normally assumes that a paragraph is preceded by a space—one line in **nroff**, and one-half line in **troff**, and the first word is indented. These rules can be changed if desired, but they are changed by changing the interpretation of **.P**, not by retyping the document.

To actually produce a document in standard format using **−mm**, use the command

```
troff −mm files ...
```

for the typesetter and

nroff −mm files ...

for a terminal. The **−mm** argument tells **troff** and **nroff** to use the manuscript package of formatting requests. There are several similar packages; check with a local expert to determine which ones are in common use on your machine. The proper terminal filter for the terminal should be used in the command line. The terminal filter option is indicated by **−T** followed by the terminal type. The terminal types are known by various XELOS system utility calls found in Section 1 of the *XELOS User Reference Manual.*

Supporting Tools

In addition to the basic formatters, there is a host of supporting programs that help with document preparation. The list in the next few paragraphs is far from complete, so browse through the *XELOS User Reference Manual* and check with XELOS operating system users for other possibilities.

Both **eqn(1)** and **neqn** (see **eqn** for more information) programs let you integrate mathematics into the text of a document in an easy-to-learn language that closely resembles the way you would speak it aloud.

For example, the **eqn** input

sum from i=0 to n x sub i ˜=˜ pi over 2

produces the output

$$\sum_{i=0}^{n} x_i = \frac{\pi}{2}$$

The program **tbl(1)** provides an analogous service for preparing tables.

The **tbl** program does all the computations necessary to align complicated columns with elements of varying widths.

The **spell**(1) program detects possible spelling mistakes in a document. The **spell** program compares the words in your document to a dictionary (stored in memory) and prints those words that are not in the dictionary. It knows enough about English spelling to detect plurals and the like, so it does a good job.

The **grep**(1) program looks through a set of files for lines that contain a particular text pattern (rather like the editor's context search does, but on a bunch of files). For example,

```
grep 'ing$' chap*
```

will find all lines that end with the letters **ing** in the files *chap**. The "$" indicated that the pattern to search for is at the end of the line, whereas a "^" indicates that the pattern to search for is at the beginning of a line. (It is almost always a good practice to put single quotes around the pattern to be searched for in case it contains characters like * or $ that have a special meaning to the shell.) The **grep** program is often used to locate the misspelled words detected by the **spell** program.

The **diff**(1) program prints a list of the differences between two files, so that two versions of something can automatically be compared. This is a vast improvement over proofreading by hand.

The **wc**(1) program counts the words, lines, and characters in a set of files. The **tr**(1) program translates characters into other characters. For example, **tr** will convert upper-case to lower-case and vice versa. This translates upper-case into lower-case:

```
tr [A-Z] [a-z] <input >output
```

The **sort**(1) program sorts files in a variety of ways while **cxref**(1) makes cross-references. The **ptx**(1) program makes a permuted index (keyword-in-context listing). The **sed**(1) program provides many of the editing

facilities of **ed** but can apply them to arbitrarily long inputs. The **awk**(1) program provides the ability to do both pattern matching and numeric computations and to conveniently process fields within lines. These programs are for more advanced users and are not limited to document preparation. Put them on your list of things to learn.

Most of these programs are either independently documented in the supplemental package like **eqn**(1) and **tbl**(1) in the DOCUMENTER'S WORKBENCH software option, or the programs are sufficiently simple enough so that the description in the *XELOS User Reference Manual* is an adequate explanation.

Hints for Preparing Documents

Most documents go through several versions (always more than expected) before they are finally finished. Accordingly, you should do whatever possible to make the job of changing them easy.

First, when you do the purely mechanical operations of typing, type so that subsequent editing will be easy. Start each sentence on a new line. Make lines short, and break lines at natural places, such as after commas and semicolons, rather than randomly. Since most people change documents by rewriting phrases and adding, deleting, and rearranging sentences, these precautions simplify any editing needed later.

Keep the individual files of a document down to modest size, perhaps 10 to 15 thousand characters. Larger files edit more slowly. If a dumb mistake is made, it is better to clobber a small file than a big one. Split the files at natural boundaries in the document for the same reasons that you start each sentence on a new line.

The second aspect of making changes to documents easy is not to commit to the formatting details too early. One of the advantages of formatting packages is permitting format decisions to be delayed until the last possible moment. Indeed, until a document is printed, it is not even decided whether it will be typeset or printed out on a line printer.

BASICS FOR BEGINNERS

As a rule of thumb, a document should be produced in terms of a set of requests or commands (macros) for all but the most trivial jobs. The macros used should then be defined either by using one of the existing macro packages (the recommended way) or by defining your own **nroff** and/or **troff** macros. As long as the text is entered in some systematic way, it can always be cleaned up and formatted by a judicious combination of editing commands and macro definitions.

Programming

There will be no attempt made to teach any of the programming languages available, but a few words of advice are in order. One of the reasons why the XELOS operating system is a productive programming environment is that there is already a rich set of tools available. Facilities like pipes, I/O redirection, and the capabilities of the shell often make it possible to do a job by pasting together programs that already exist instead of writing a program completely from scratch.

The *XELOS Programmer Reference Manual* contains the XELOS system programming utilities.

Shell Programming

The pipe mechanism lets you fabricate quite complicated operations out of spare parts that already exist. For example, the first draft of the **spell** program was (roughly)

cat ...	*collect the files*
| tr ...	*put each word on a new line*

|tr ...	*delete punctuation, etc.*
|sort	*into dictionary order*
|uniq	*discard duplicates*
|comm	*print words in text but not in dictionary*

More pieces have been added subsequently, but this goes a long way for such a small effort.

The editor can be made to do things that would normally require special programs on other systems. For example, to list the first and last lines of each of a set of files, such as a book, the user could laboriously type:

```
ed
e chap1.1
1p
$p
e chap1.2
1p
$p
etc.
```

The same job can be performed much more easily. One procedure is to type

```
ls chap* >temp
```

to get the list of file names into a file called *temp*. The *temp* file is then edited using global commands as follows:

```
1,$ s/^.*$/e &\
1p\
$p/
```

The results are written into the *script* file (1,$ w script) and then the following command is entered:

```
ed <script
```

This will produce the same output as the laborious hand typing. Another method is using shell loops to repeat a set of commands over and over again for a set of arguments as illustrated below:

```
for i in chap*
do
        ed $i <script
done
```

This sets the shell variable *i* to each file name in turn, then does the command. This command can be entered at the terminal or put in a file for later execution. Before the file can be executed, it may be necessary to change the mode by entering the following:

```
chmod +x filename
```

Programming with Shell

An option often overlooked by new users is that the **shell** is itself a programming language, with variables, control flow **if-else**, **while**, **for**, **case**, subroutines, and interrupt handling. Since there are many building-block programs, the user can sometimes avoid writing a new program merely by piecing together some of the building blocks with shell

command files.

We will not go into any details here; examples and rules can be found in section "AN INTRODUCTION TO SHELL" described later in this volume.

Programming in C

The C language is a reasonable choice of a programming language when undertaking anything substantial. Everything in the XELOS operating system is based on the C language. The system itself is written in C, as are most of the programs that run on the system. The C language is also an easy language to use once you get started. The C language is introduced and fully described in *The C Programming Language* by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). Several sections of the manual describe the system interfaces, that is, how to do I/O and similar functions.

Most input and output in C is best handled with the standard I/O library, which provides a set of I/O functions that exist in compatible form on most machines that have C compilers. In general, it's wisest to confine the system interactions in a program to the facilities provided by this library. (Refer to Section 3 of the *XELOS User Reference Manual.)*

The C programs that do not depend too much on the special features of the XELOS operating system (such as pipes) can be moved to other computers that have C compilers.

There are a number of supporting programs that go with C. The **lint**(1) program checks C programs for potential portability problems and detects errors such as mismatched argument types and uninitialized variables.

For larger programs (anything whose source is on more than one file), the **make**(1) program allows you to specify the dependencies among the source files and the processing steps needed to make a new version. The program then checks the times that the pieces were last changed and does

the minimal amount of recompiling to create a consistent updated version.

The debugger **sdb**(1) program is useful for digging through the dead bodies of C programs but is rather hard to learn to use effectively. The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

The C compiler provides a limited statistical service, so a user can find where programs spend their time executing and what parts of a program are worth optimizing. Compile the programs with the **−p** option; after the test run, use the **prof**(1) command to print a program execution profile. The command **time**(1) will give the gross run-time statistics of a program, but the times are not very accurate or reproducible.

Other Languages

If FORTRAN *must* be used, there are two possibilities— FORTRAN 77 and ratfor. The user might consider **ratfor** which provides decent control structures and free-form input that characterize C, yet permits the writing of code that is also portable to other environments. Bear in mind that XELOS operating system FORTRAN tends to produce large and relatively slow-running programs. Furthermore, supporting software like **prof**(1), etc., are all virtually useless with FORTRAN programs. If there is a FORTRAN 77 compiler on your system, it may be a viable alternative to **ratfor** and has the nontrivial advantage that it is compatible with the C language and related programs. (The ratfor processor and C tools can be used with FORTRAN 77, too.)

If your application requires translating a language into a set of actions or another language, the user is, in effect, building a compiler, though probably a small one. In that case, the **yacc**(1) compiler-compiler is recommended for use, which aids in developing a compiler quickly.

The **lex**(1) lexical analyzer generator does the same job for the simpler languages that can be expressed as regular expressions. It can be used by itself or as a front-end to recognize inputs for a **yacc-based** program.

Both **yacc** and **lex** require some sophistication to use, but the initial effort of learning them can be repaid many times over in programs that are easy to change later.

Chapter 4

TUTORIAL – TEXT EDITOR

PAGE

INTRODUCTION 1

GENERAL 1

GETTING STARTED 2

EXERCISES – TRY THEM! 7

EXERCISE 1 7

EXERCISE 2 10

EXERCISE 3 13

EXERCISE 4 17

EXERCISE 5 21

EXERCISE 6 25

EXERCISE 7 28

THE GLOBAL COMMANDS 30

SPECIAL CHARACTERS 31

SUMMARY OF COMMANDS AND LINE NUMBERS 36

Chapter 4

TUTORIAL – TEXT EDITOR

INTRODUCTION

Almost all text input on the XELOS operating system is done with the standard text editor **ed**(1). This is a tutorial guide to help beginners get started with text editing.

Although this guide does not cover everything about the XELOS operating system, it does discuss enough for most user's day-to-day needs. This includes printing, appending, changing, deleting, moving, and inserting entire lines of text; reading and writing files; context searching and line addressing; substituting; global changing; and using some special characters for easier editing.

Throughout this section, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *XELOS Administrator Reference Manual*. Other references to entries of the form **name**(N), where "N" is a number (1 or 6) possibly followed by a letter, refer to entry **name** in section N of the *XELOS User Reference Manual*. Entries where "N" is a number (2 through 5) possibly followed by a letter, refer to entry **name** in section N of the *XELOS Programmer Reference Manual*.

GENERAL

The **ed** program is a "text editor", that is, an interactive program for creating and modifying "text" using directions (commands) provided by a user at a terminal. The text is often a document like this one or perhaps data for a program.

This tutorial is meant to simplify learning **ed**. The recommended way to learn **ed** is to read this document, while simultaneously using **ed** to follow

the examples, then to read the description in Section 1 of the *XELOS User Reference Manual*. Getting advice from experienced XELOS operating system users and experimenting with **ed** are also useful.

Do the exercises! The exercises illustrate techniques not completely discussed in the actual text. A summary at the end of this section summarizes the commands.

Disclaimer

This is a tutorial introduction and guide only. For this reason, no attempt is made to cover more than a part of the facilities that **ed** offers (although this fraction includes the most useful and frequently used facilities). Also, there is not enough space to explain the basic XELOS operating system procedures. It is assumed that the user knows how to log on to the XELOS operating system and has a vague understanding of what a XELOS operating system file is. For more on the XELOS operating system facilities, refer to the section, "Basics For Beginners".

The user must also know what character to type as the end of line character on the user's particular terminal. This character is the RETURN or newline character (key) on most terminals. Hereafter, reference to the end of line character, whatever it is, will be referred to as RETURN.

GETTING STARTED

Assume that the user has logged in to a XELOS operating system and it has just printed the *prompt character*, usually a

```
$
```

The easiest way to invoke **ed** is to type:

ed (followed by a RETURN)

You (the user) are now ready to go. The **ed** program is waiting to be told what to do.

Creating Text – The Append Command "a"

As your first problem, suppose some text is to be created starting from scratch. Perhaps the very first draft of a document or paper is to be entered. Normally, it will have to start somewhere and undergo modifications (editing) later. This part will describe how to enter some text to get a file of text started. How to make changes and corrections to the text is described later.

When **ed** is first invoked, it is rather like working with a blank piece of paper (the file) – there is no text or information present on the paper (in the file). The text must be supplied by the person using **ed**; it is usually done by typing in the text or by reading it into **ed** from a file. We will start by typing in some text and return shortly to how to read files.

First a bit of terminology. In **ed** jargon, the text being worked on is said to be "kept in a buffer." Think of the buffer as a work space, if desired, or simply as the information that is to be edited. In effect, the buffer is like the piece of paper on which we will write things, then change some of them, and finally file the whole thing away for another day.

The user tells **ed** what to do to the text by typing instructions called "commands." Most commands consist of a single lower-case letter. Each command is typed on a separate line. (Sometimes the command is preceded by information about the line or lines of text to be affected – these will be described below.) The **ed** text editor makes no response to most commands – there is no prompting or response messages like "ready". (This silence is preferred by experienced users). The first command is append, written as the letter

```
a
```

on a command line all by itself. It means "append (or add) text lines to the buffer as I type them in."

Appending is rather like writing fresh material on a piece of paper. So to enter lines of text into the buffer, just type an

```
a
```

followed by a RETURN and the lines of text, like this:

```
a
Now is the time
for all good men
to come to the aid of their party.
.
```

The only way to stop appending is to type a line that contains only a period. The "." is used to tell **ed** that the appending is finished. (Even experienced users forget to terminate appending with a "." sometimes. If **ed** seems to be ignoring your entries, type an extra line with just the "." on it. You may then find you have added some garbage lines to your text, which you will have to take out later.)

After the append command has been used, the buffer will contain the following three lines:

```
Now is the time
for all good men
to come to the aid of their party.
```

The **a** and the "." are not there because they are not text.

To add more text to what already exists, just issue another **a** command and continue typing.

Error Messages (?)

If at any time the user makes an error in the commands typed into **ed**, the text editor will tell the user by typing the following:

```
?
```

This is about as cryptic as it can be, but with practice, the user can usually figure out the goof. The user can get a brief explanation of the error by typing

```
h
```

The **help** command gives a short error message that explains the reason for the most recent **?** diagnostic.

Writing Text File – The Write Command "w"

It is likely that you will want to save your text for later use. To write out the contents of the buffer onto a file, use the write command

```
w
```

followed by the file name to write on. This will copy the buffer's contents onto the specified file (destroying any previous information on the file). To save (write) the text in a file named *junk*, for example, type:

```
w junk
```

Leave a space between **w** and the file name. The **ed** program will respond by printing the number of characters it wrote out. In this case, **ed** would respond with:

```
68
```

(Remember that blanks and the return character at the end of each line are included in the character count.) Writing a file just makes a copy of the text – the buffer's contents are not disturbed, so the user can go on adding lines to it. This is an important point. The **ed** program at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a **w** command. (Writing out the text onto a file from time to time as it is being created is a good idea. If the system crashes or if the user makes some horrible mistake, all the text in the buffer will be lost but any text that was written onto a file is relatively safe.)

Leaving ed – The Quit Command "q"

To terminate a session with **ed**, first save your text by writing it onto a file using the **w** (write) command, and then type the **q** (quit) command:

```
q
```

The system will respond with the prompt character:

```
$
```

At this point your buffer vanishes, with all its text, which is why the user

would want to write before quitting. Actually **ed** will print the character

```
?
```

if the user tries to quit without writing. At this point, the user writes if desired; if not, another **q** will get you out regardless and will not save the text in the buffer.

EXERCISES – TRY THEM!

EXERCISE 1

Enter **ed** and create some text using the append command **a**

```
a
...text...
.
```

Note that no system prompt appears while in the text editor. Do not forget to write the text into memory with the write command **w**. Write it into memory using the **w** command. Then leave **ed** with the **q** command and print the file to see that everything worked. To print a file, enter

```
pr filename
   or
cat filename
```

in response to the prompt character ($). Try both.

Reading Text File – The Edit Command "e"

A common way to get text into the buffer is to read it from another file in the file system. This is what you do to edit text that you saved with the **w** command in a previous session. The edit command

```
e
```

retrieves the entire contents of a file into the buffer.

So if the user had saved the three lines "Now is the time", etc., with a **w** command in an earlier session, the edit command

```
e junk
```

would place the entire contents of the file *junk* into the buffer and respond with a number

```
68
```

which is the number of characters in the file *junk*. *If anything was already in the buffer, it is deleted first.*

If the **e** command is used to read a file into the buffer, then the user does not need to use a file name after a subsequent **w** command; **ed** remembers the last file name used in an **e** command, and **w** will write on this file. Thus a good practice to follow is:

```
ed
e filename
[editing session]
.
w
q
```

This way, the user can simply enter **w** from time to time and be secure in the knowledge that if the user got the file name right at the beginning, the user is writing into the proper file each time. Note that after each edit command **e** or each write command **w** the number of characters is returned by **ed**. The user can find out at any time what file name **ed** is remembering by typing the file command **f**. In this example, if you typed

```
f
```

ed would reply

```
junk
```

Reading Text – The Read Command "r"

Sometimes you want to read a file into the buffer without destroying anything that is already in the buffer. This is done by the read command **r**. The command

```
r junk
```

will read the file *junk* into the buffer. The command appends the file specified to the end of whatever file is already in the buffer. So if you do a read after an edit command such as

```
e junk
r junk
```

the buffer will contain *two* copies of the original text as follows:

```
Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.
```

Like the **w** and **e** commands, **r** prints the number of characters read in after the reading operation is complete. Generally speaking, **r** is much less used than **e**.

The read command **r** may also be used to read a file external to the buffer into the file in the buffer. While in **ed** and at the current line, enter the command

```
.r filename
```

and filename will be read into the file (already in the buffer) immediately after the current line. None of the file in the buffer is destroyed, rather the external file *filename* has been read into and been combined with the file already in the buffer. The file that was read remains in filename also. You only copied it. Notice the difference between "r" and ".r".

EXERCISE 2

Experiment with the **e** command – try reading and printing various files. The user may get an error **?name** where *name* is the name of a file. This means that the file does not exist. Some typical causes of getting an empty file are spelling the file name wrong or perhaps trying to read or write a particular file which your permissions will not allow. Try alternately reading and appending to see that they work similarly. Verify that

```
ed filename
```

is exactly equivalent to

```
ed
e filename
```

What does

```
f filename
```

do?

Printing Buffer Contents – Print Command "p"

To print or list the contents of the buffer (or parts of it) on the terminal, use the print command **p**. This is done as follows. Specify the line numbers where printing is to begin and end. These numbers have a comma between the beginning number and the ending number, i.e., "beginning line number, ending line number p". Thus, to print the first ten lines of the contents of any buffer (i.e., lines 1 through 10), type:

```
1,10p     (prints lines 1 through 10)
```

The **ed** will respond by printing the specified starting line (1) through the specified ending line (10).

Suppose it is desirable to print *all* the lines in the buffer. You could use "1,30p" as above if it is known there were exactly 30 lines in the buffer. But in general, it is not known how many lines there are, so what can be used for the ending line number? The **ed** program provides a shorthand

symbol for "line number of the *last line* in the buffer" – the dollar sign **$**. To print all the lines in the buffer, use it this way:

```
1,$p    (Prints all lines in buffer)
             or
,p      (Prints all lines in buffer also)
```

This will print *all* the lines in the buffer (line 1 through the last line). The "1,$p" can be abbreviated "$,p". To stop the printing before the last line is printed, push the **DEL** key or the **DELETE** (or equivalent) key on the terminal. The **ed** program will respond

```
?
```

and wait for the next input command.

To print the *last* line of the buffer, you could use

```
$,$p
```

but **ed** lets you abbreviate this to

```
$p
```

Any *single* line can be printed by typing the line number followed by a **p**. Thus

```
1p
```

produces the response

```
Now is the time
```

which is the first line of the buffer.

In fact, **ed** lets you abbreviate even further. You can print any single line by typing *just* the line number – no need to type the letter **p**. So by entering

```
$
```

ed will print the last line of the buffer. Entering a single line number will print that line only.

It is also possible to use **$** in combinations like

```
$-5,$p
```

which prints the *last five* lines of the buffer. This helps to determine the end of the contents of the buffer when more is to be entered.

EXERCISE 3

Create some text using the **a** command and experiment with the **p** command. The user will find, for example, that line 0 or a line beyond the end (last line) of the buffer cannot be printed. Attempts to print a buffer in reverse order by entering

```
3,1p
```

will *not* work.

The Current Line "." or Dot

Suppose the buffer still contains the six lines of text (as in Exercise 1), and the following was entered

```
1,3p
```

and **ed** has printed the three lines.

Try typing just

```
p       (no line numbers)
```

This will print

```
to come to the aid of their party.
```

which is the third line of the buffer. In fact it is the last (most recent) line that anything was done to. (The line just printed!) The **p** command can be repeated without line numbers, and it will continue to print line 3.

The reason is that **ed** maintains a record of the last line that anything was done to (in this case, line 3, which was just printed) so that it can be used instead of an explicit line number. This most recent line is referred to by the shorthand symbol

```
.       (Pronounced "dot")
```

Dot is a line number in the same way that **$** is. Dot means exactly "the current line", or loosely, "the line something was done to most recently." The dot can be used in several ways – one possibility is to enter:

```
.,$p
```

This will print all the lines from (including) the current line to the end (last line) of the buffer. In our example, these are lines 3 through 6.

Some commands change the value of dot, while others do not. The print command **p** sets dot to the number of the last line printed; the last command entered (.,$p) will set both "." and **$** to the last line in the buffer (line 6).

Dot is most useful when used in combinations like:

```
.+1     (or equivalently, .+1p)
```

This means "print the next line" and is a handy way to step slowly through a buffer. The user can also enter

```
.-1     (or .-1p)
```

which means "print the line *before* the current line." This enables stepping through the buffer backwards if desired. Another useful one is something like

```
.-3,.-1p
```

which prints the previous three lines.

Do not forget that all of these change the value of dot. The user can find out what dot is at any time by typing

```
.=      (dot line number is ?)
```

The **ed** program will respond by printing the value (line number) of dot.

Let us summarize some things about the **p** command and dot. Essentially, **p** can be preceded by 0, 1, or 2 line numbers (for our example). If there is no line number given, it prints the "current line", the line that dot

refers to. If there is one line number given with or without the letter **p**, it prints that line and sets dot there. If there are two line numbers separated by a comma, it prints all the lines in that range from the first number to the last number, and sets dot to the last line printed. If two line numbers are specified, the first cannot be bigger than the second (refer to the beginning of "EXERCISE 3").

Typing a single RETURN will cause printing of the next line – it is equivalent to:

```
.+1p
```

Try it. Typing a ^ is equivalent to typing the minus –. It can be used in multiples, as ^^^, which will move the current line or dot line backwards three lines from the current line. The "–" or the "^" can be considered equivalent to "–1p" since either moves the dot back one line.

Deleting Lines – The Delete Command "d"

Suppose three extra lines in the buffer are not needed. They may be removed by use of the delete command:

```
d
```

Except that **d** deletes lines instead of printing them, its action is similar to that of the print command **p**. The lines to be deleted are specified for **d** exactly as they are for **p** as follows:

```
starting line, ending line d
```

Thus the command

```
4,$d
```

deletes lines 4 through the end. There are now three lines left that can be checked by using:

```
1,$p
```

And notice that **$** now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to **$**.

The delete command **d** and the print command **p** may be used together, thus

```
dp
```

which deletes the current line, prints the following line, and sets dot to the line printed.

EXERCISE 4

Experiment with **a**, **e**, **r**, **w**, **p**, and **d** until you become familiar with their use. While experimenting, also use "dot", **$**, and line numbers to understand their use.

When you start to feel adventurous, try using line numbers with **a**, **r**, and **w** as well. The user will find that **a** will append lines *after* the line number that you specify (rather than after dot); **r** reads a file in *after* the line number you specify (not necessarily at the end of the buffer); and **w** will write out exactly the lines specified, not necessarily the whole buffer. These variations are sometimes handy. For instance, a file can be inserted at the beginning of a buffer by entering:

```
0r filename
```

Lines can be entered at the beginning of the buffer by using:

```
0a
...text...
.
```

Notice that ".w" is *very* different from

```
.
w
```

Modifying Text – The Substitute Command "s"

We are now ready to try one of the most important of all commands – the substitute command

```
s
```

This is the command that is used to change individual words or letters within a line or group of lines. The substitute command is used for correcting spelling mistakes and typing errors.

Suppose that, because of a typing error, line 1 says

```
Now is th time
```

notice the *e* has been left off. The **s** command can be used to fix this as follows:

```
1s/th/the/
```

This says: in line 1, substitute for the characters *th* the characters *the*. Since **ed** will not print the result automatically, enter

```
p
```

to verify that the substitution worked, and you should get

```
Now is the time
```

which is what is desired. Notice that dot must have been set to the line where the substitution took place since the **p** command printed that line. Dot is always set this way with the **s** command.

The general way to use the substitute command is

```
starting-line, ending-line s/change this/to this/
```

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in *all* the lines between *starting–line* and *ending–line*. Only the *first* occurrence on *each* line is changed however. If *every* occurrence is to be changed, see "EXERCISE 5". The rules for line numbers are the same as those for the print command **p** except that dot is set to the last line changed. (But there is a trap for the unwary: if no substitution took place, dot is *not* changed. This causes an error response **?** as a warning.)

Thus, the following can be entered

```
1,$s/speling/spelling/
```

to correct the first spelling mistake (speling in this case) on each line in the

text. (This is useful for people who are consistent misspellers!)

If no line numbers are given, the **s** command assumes we mean "make the substitution on line dot", so it changes things only on the current line. This leads to the very common sequence

```
s/something/something else/p
```

which makes some correction on the current line and then prints it (current line) to make sure it worked out right. If it did not, you can try again. Notice that there is a **p** on the same line as the **s** command. With few exceptions, **p** can follow any substitute command.

It is also legal to say

```
s/....//
```

which means change the first string of characters (....) to *nothing*, i.e., remove them. This is useful for deleting extra words in a line or removing extra letters from words.

For instance, if the buffer contained

```
Nowxx is the time
```

this can be corrected by entering

```
s/xx//p
```

to get

```
Now is the time
```

Notice that // (two adjacent slashes) means "no characters" *not* a blank. There *is* a difference! (See "Context Searching" under "EXERCISE 5" for another meaning of "//").

EXERCISE 5

Experiment with the substitute command. See what happens if you substitute for some word on a line with several occurrences of that word. For example, enter

```
a
the other side of the coin
.
s/the/on the/p
```

which results in the following:

```
on the other side of the coin
```

A substitute command changes only the first occurrence of the first string.

All occurrences can be changed by adding a **g** (for "global") command to the **s** command, like this:

```
s/.../.../gp
```

Try other characters instead of slashes to delimit the two sets of characters in the **s** command – anything should work except blanks or tabs.

If strange results are produced by inputing

^ . $ [* \ &

read the part under "Special Characters" in this section.

Context Searching "/...../"

When the substitute command is mastered, you may move on to another highly important feature of **ed**(1) – context searching.

Suppose the original three lines of text in the buffer is as follows:

```
Now is the time
for all good men
to come to the aid of their party.
```

Suppose the word *their* is to be changed to *the*. How is the line that contains *their* located? With only three lines in the buffer, it is pretty easy to keep track of what line the word *their* is on. But if the buffer contained several hundred lines and you had been making changes, deleting and rearranging lines, etc., you would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context (unique text) on it.

The way to say "search for a line that contains this particular string of characters" or "unique text" is to type:

```
/string of characters to find/
```

For example, the **ed** expression

```
/their/
```

is a context search which is sufficient to find the desired line – it will locate the next occurrence of the characters between slashes ("their"). It also sets dot to that line and prints that line for verification:

```
to come to the aid of their party.
```

"Next occurrence" means that **ed** starts looking for the string at line ".+1" and searches to the end of the buffer, then continues at line 1 and searches to line dot. That is, the search "wraps around" from **$** to **1**. It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters cannot be found in any line, **ed** types the error message

```
?
```

Otherwise, it prints the line it found.

The search for the desired line *and* the substitution can be done together, like this

```
/their/s/their/the/p
```

which will yield

```
to come to the aid of the party.
```

There were three parts to that last command: context search for the desired line, make the substitution, and print the line.

The expression "/their/" is a context search expression. In the simplest form, all context search expressions are like this – a string of characters

surrounded by slashes. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line or as line numbers for some other command, like **s**. They were used both ways in the examples above.

Suppose the buffer contains the three familiar lines

```
Now is the time
for all good men
to come to the aid of their party.
```

Then the **ed** line numbers

```
/Now/+1
/good/
/party/-1
```

are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, enter

```
/Now/+1s/good/bad/
     or
/good/s/good/bad/
     or
/party/-1s/good/bad/
```

The choice is dictated only by convenience. All three lines could be printed by entering

```
/Now/,/party/p
     or
/Now/,/Now/+2p
```

or by any number of similar combinations. The first one of these might be better if you do not know how many lines are involved. The basic rule is: a context search expression is the *same* as a line number, so it can be used wherever a line number is needed.

EXERCISE 6

Experiment with context searching. Try a body of text with several occurrences of the same string of characters and scan through it using the same context search.

Try using context searches as line numbers for the substitute, print, and delete commands. They can also be used with **r**, **w**, and **a**.

Try context searching using "?text?" instead of "/text/" This scans lines in the buffer in reverse order rather than normal (forward order). This is sometimes useful if you go too far while looking for some string of characters – it is an easy way to back up.

If funny results are obtained with any of the characters

```
^   .   $   [   *   \   &
```

read the part in this section on "Special Characters".

The **ed** program provides a short method for repeating a context search for the same string. For example, the **ed** line number

```
/string/
```

will find the next occurrence of "string". It often happens that this is not the desired line, so the search must be repeated. This can be done by typing merely:

```
//
```

This short method stands for "the most recently (last) used context search expression". It can also be used as the first string of the substitute command, as in

```
/string1/s//string2/
```

which will find the next occurrence of **string1** and replace it by **string2**. This can save a lot of typing. Similarly

```
??
```

means "scan backwards for the same expression."

Change and Insert Commands "c" and "i"

This section discusses the change command

```
c
```

which is used to change the current line or to replace the current line with a group of one or more lines, and the insert command

```
i
```

which is used for inserting a group of one or more lines immediately before the current line.

"Change", written as

```
c
```

is used to replace a number of lines with different lines, which are typed in at the terminal. For example, to change the first line (.+1) past the current line through the last line ($) of a file to something else, type

```
.+1,$c
...type the lines of text you want here...
.
```

The lines typed between the **c** command and the '**.**' (dot) command will take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines which have errors.

If only one line is specified in the **c** command, then just that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of '**.**' (dot) to end the input – this works just like the '**.**' (dot) in the **a** command and must appear by itself at the beginning of a new line. If no line number is given, line dot is replaced. The value of dot is set to the last line you typed in.

"Insert" is similar to append – for instance

```
/string/i
...type the lines to be inserted here...
.
```

will insert the given text *before* the next line that contains "string". The text between **i** and the '**.**' (dot) is inserted *before* the specified line. If no line number is specified, the dot line is used. Dot is set to the last line inserted.

EXERCISE 7

"Change" is rather like a combination of delete followed by insert. Experiment to verify that

```
starting-line,ending-line d
i
...text...
.
```

is almost the same as

```
starting-line,ending-line c
...text...
.
```

These are not *precisely* the same if the last line ($) gets deleted. Check this out. What is dot?

Experiment with the append command **a** and the insert command **i** to see that they are similar but not the same. You will observe that

```
line-number a
...text...
.
```

appends *after* the given line, while

```
line-number i
...text...
.
```

inserts *before* it. Observe that if no line number is given, **i** inserts before line dot, **a** appends after line dot, and **c** changes line dot.

Moving Text Around – The Move Command "m"

The move command **m** is used for cutting and pasting – it allows a group of lines to be moved from one place to another in the buffer. Suppose the first three lines of the buffer are to be placed at the end of the buffer instead of at the beginning. This could be performed by entering:

```
1,3w temp
$r temp
1,3d
```

(Do you see why?) This method will work, but it is a lot easier using the **m** command as follows:

```
1,3m$
```

The general case is:

```
starting-line,ending-line m after this line
```

Notice that there is a third line to be specified – the line after which the other lines are to be moved. Of course, the lines to be moved can be specified by context searches; if you had

```
First paragraph
...
end of first paragraph.
Second paragraph
...
end of second paragraph.
```

the two paragraphs could be reversed like this:

```
/Second/,/end of second/m/First/-1
```

Notice the "-1" – the moved text goes *after* the line mentioned. Dot gets set to the last line moved.

THE GLOBAL COMMANDS

The two global commands are **g** and **v**. The global command **g** is used to execute one or more **ed** commands on all those lines in the buffer that match some specified string. For example

```
g/peling/p
```

prints all lines that contain "peling". More usefully,

```
g/peling/s//pelling/gp
```

makes the substitution everywhere on the line, then prints each corrected line.

Compare this to

```
1,$s/peling/pelling/gp
```

which only prints the last line substituted. Another subtle difference is that the **g** command does not give a **?** – if "peling" is not found, where the **s** command will.

There may be several commands used in conjunction with the **g** command, but every line except the last must end with a backslash "\". For example:

```
g/xxx/-1s/abc/def\
.+2s/ghi/jkl\
.-2,.p
```

makes changes in the lines before and after each line that contains "xxx", then prints all three lines.

The **v** command is the same as **g** except that the commands are executed on every line that does *not* match the string following **v**. The following input

```
v/ /d
```

deletes every line that does not contain a blank.

SPECIAL CHARACTERS

You may have noticed that things just did not work right when you used some characters like **.**, *****, **$**, and others in context searches and in the s command. The reason is rather complex, although the cure is simple. Basically, **ed** treats these characters as special, with special meanings. For instance, ***in a context search or the first string of the substitute command only***,

```
/x.y/
```

means "a line with an **x**, *any character*, and a **y**", *not* just "a line with an **x**, a period, and a **y**."

The following is a complete list of the special characters that can cause trouble:

```
^ . $ [ * \ &
```

Warning: The backslash character "\" is special to "ed". For safety's sake, avoid it where possible.

If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus

```
s/\\\.\*/backslash dot star/
```

will change "\.*" into "backslash dot star".

Here is a brief synopsis of the other special characters. First, the circumflex "^" signifies the beginning of a line. Thus

```
/^string/
```

finds "string" only if it is at the beginning of a line. It will find

```
string
```

but not

```
the string...
```

The dollar sign "$" is just the opposite of the circumflex; it means the end of a line.

The input

```
/string$/
```

will only find an occurrence of "string" at the end of some line. This implies, of course, that

```
/^string$/
```

will find only a line that contains just "string" and

```
/^.$/
```

finds a line containing exactly one character.

The character ".", as we mentioned above, matches anything. For example, the input

```
/x.y/
```

matches any of the following:

```
x+y
x-y
x y
x.y
```

This is useful in conjunction with "*" which is a repetition character. The "a*" is a shorthand input for "any number of a's" therefore ".*" matches any number of anythings.

For example, input

```
s/.*/stuff/
```

which changes an entire line, or

```
s/.*,//
```

which deletes all characters in the line up to and including the last comma. (Since ".*" finds the longest possible match, this goes up to the last comma.)

The "[" is used with the "]" to form *character classes*; for example,

```
/[0123456789]/
```

matches any single digit – any one of the characters inside the brackets will cause a match. This can be abbreviated to

```
[0-9]
```

Finally, the "&" is another *shorthand character* – it is used only on the right-hand part of a substitute command where it means "whatever was matched on the left-hand side". It is used to save typing.

Suppose the current line contained

```
Now is the time
```

and you wanted to put parentheses around it. One tedious method is just to retype the line. Another method is to enter

```
s/^/(/
s/$/)/
```

using your knowledge of "^" and "$". But the easiest way uses the "&" as follows:

```
s/.*/(&)/
```

This says "match the whole line and replace it by itself surrounded by parentheses." The "&" can be used several times in a line; consider using

```
s/.*/&? &!!/
```

to produce

```
Now is the time? Now is the time!!
```

You do not have to match the whole line, of course. If the buffer contains

```
the end of the world
```

you could type

```
/world/s//& is at hand/
```

to produce

```
the end of the world is at hand
```

Observe this expression carefully, for it illustrates how to take advantage of **ed** to save typing. The string "/world/" found the desired line; the shorthand "//" found the same word in the line; and the "&" saves you from typing it again.

The "&" is a special character only within the replacement text of a substitute command and has no special meaning elsewhere. You can *turn off* the *special meaning* of "&" by preceding it with a backslash "\".

Inputing

s/ampersand/\&/

will convert the word "ampersand" into the literal symbol "&" in the current (dot) line.

SUMMARY OF COMMANDS AND LINE NUMBERS

The general form of the **ed** text editor commands is the command **name**, perhaps preceded by one or two line numbers. In the case of the edit command **e**, the read command **r**, and the write command **w**, the command **name** is also followed by a *file name*. Normally, only one command is allowed to be entered per line, but a print command **p** may follow any other command (except for the edit command **e**, the read command **r**, the write command **w**, and the quit command **q**).

a *Append*, adds lines to the buffer (at line dot, unless a different line is specified). Appending continues until a dot "." is typed at the beginning (first character) of a new line. Dot is set to the last line appended.

c *Change* the specified lines to the new text which follows. Entering new lines is terminated by a dot "." as with **a**. If no lines are specified, the current line (dot) is replaced. Dot is set to last line changed.

d *Delete* the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line unless **$** is specified, in which case dot is set to the last line, **$**.

e — *Edit* new file. Any previous contents of the buffer are thrown away, so issue a write command **w** beforehand.

f — Print the remembered *file* name. If a name follows **f**, the remembered name will be set to it.

g — The *global* command **g/---/commands** will execute the commands on those lines that contain "---".

i — *Insert* lines before the specified line or the current line (dot line) until a "." is typed at the beginning of a new line. Dot is set to last line inserted.

m — *Move* lines specified to the line named after **m**. Dot is set to the last line moved.

n — Print the *number* of the addressed line(s) followed by a tab and the line itself.

p — *Print* specified lines. If none specified, print line dot. A single line number is equivalent to "line number". A single RETURN prints the next line, i.e., the dot plus one line, ".+1".

q — The *quit* command exits from **ed**. It wipes out all text in buffer if you give it twice in a row without first giving a write command **w**.

r — *Read* a file into buffer (at end unless specified elsewhere). Dot set to last line read. If **.r filename** is used, the filename is read into the buffer immediately after the dot line.

s — The **s/string1/string2/** command is used to *substitute* the characters "string1" into "string2" in the specified lines. If no lines are specified, make the substitution in line dot. Dot is set to last line in which a substitution took place; if no substitution took place, dot is not changed. The command **s** changes only the first occurrence of "string1" on a

line; to change all occurrences on a line, type a g after the final slash.

v The *exclude* command **v/---/commands** executes commands only on those lines that do *not* contain "---".

w The *write* command writes out the buffer contents onto a file. Dot is not changed.

.= The ".=" causes the printout of the current line number. The *dot value* prints the line number of the current line (dot line). The "=" by itself prints the value of the last line in the file.

! The "!" is a *temporary escape* command. The line "command-line" causes "command-line" to be executed as a XELOS operating system command.

/-----/ The *context search* command searches for next line which contains this string of characters "----" and prints it. Dot is set to the line where string was found. Search starts at line ".=1" then wraps around from the last line "$" to line "1" and continues to dot (the current line) if necessary.

?-----? Performs *context search* in reverse direction. Starts search at the previous line ".–1", scans to line 1, wraps around to the last line "$", and scans back to the current line (dot line) if necessary.

Chapter 5

AN INTRODUCTION TO THE SHELL

PAGE

INTRODUCTION 1

SIMPLE COMMANDS 2

Background Commands 2

Input/Output (I/O) Redirection 3

Pipelines and Filters 3

File Name Generation 4

Quoting 6

Prompting by the Shell 7

The Shell and Login 7

Summary 8

SHELL PROCEDURES 8

Control Flow—for 9

Control Flow—case 11

Here Documents 13

Shell Variables 14

Test Command 17

Control Flow—while 18

Control Flow—if 19

Debugging Shell Procedures 23

The “man” Command 23

KEYWORD PARAMETERS 24

Parameter Transmission 25

Parameter Substitution 25

Command Substitution 26

Evaluation and Quoting 28

Error Handling 31

Fault Handling 32

Command Execution 35

Invoking the Shell 38

Chapter 5

AN INTRODUCTION TO THE SHELL

INTRODUCTION

The **shell** is a command programming language that provides an interface to the XELOS operating system. Its features include control-flow primitives, parameter passing, variables, and string substitution. Constructs such as **while**, **if then else**, **case**, and **for** are available. Two-way communication is possible between the **shell** and commands. String-valued parameters, typically file names or flags, may be passed to a command. A return code is set by commands that may be used to determine control-flow, and the standard output from a command may be used as **shell** input.

The **shell** can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through *pipes* can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user. Commands can be read either from the terminal or from a file which allows command procedures to be stored for later use.

The **shell** is both a command language and a programming language that provides an interface to the XELOS operating system. This volume describes, with examples, the XELOS operating system **shell**. The "Simple Commands" part of this section covers most of the everyday requirements of terminal users. Some familiarity with the XELOS operating system is an advantage when reading this section; refer to the section "BASICS FOR BEGINNERS". The "Shell Procedures" part of this section describes those features of the shell primarily intended for use within shell commands or procedures. These include the control-flow primitives and string-valued variables provided by the **shell**. A knowledge of a programming language would be helpful when reading this section. The last part, "Keyword Parameters", describes the more advanced features of the **shell**. See Table 5.A for a defined listing of grammar words used in

this section.

Throughout this section, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *XELOS Administrator Reference Manual*. Other references to entries of the form **name**(N), where "N" is a number (1 or 6) possibly followed by a letter, refer to entry **name** in section N of the *XELOS User Reference Manual*. Entries where "N" is a number (2 through 5) possibly followed by a letter, refer to entry **name** in section N of the *XELOS Programmer Reference Manual*.

SIMPLE COMMANDS

Simple commands consist of one or more words separated by blanks. The first word is the **name** of the command to be executed; any remaining words are passed as *arguments* to the command. For example,

```
who
```

is a command that prints the names of users logged in. The command

```
ls -l
```

prints a list of files in the current directory. The argument *−l* tells **ls**(1) to print status information, size, and the creation date for each file.

Background Commands

To execute a command, the **shell** normally creates a new process and waits for it to finish. A command may be run without waiting for it to finish. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file *pgm.c*. The trailing "&" is an operator that instructs the **shell** not to wait for the command to finish. To help keep track of such a process, the **shell** reports its process number

following its creation. A list of currently active processes may be obtained using the **ps**(1) command.

Input/Output (I/O) Redirection

Most commands produce output to the *standard output* that is initially connected to the terminal. This output may be directed to a file by the notation ">" thus:

```
ls -l >file
```

The notation >*file* is interpreted by the **shell** and is not passed as an argument to **ls**(1). If *file* does not exist, the **shell** creates it; otherwise, the original contents of *file* are replaced with the output from **ls**(1). Output may be appended to a file using the notation ">>" as follows:

```
ls -l >>file
```

In this case, *file* is also created if it does not already exist.

The *standard input* of a command may be taken from a file instead of the terminal by the notation "<" thus:

```
wc <file
```

The command **wc**(1) reads its standard input (in this case redirected from *file*) and prints the number of characters, words, and lines found. If only the number of lines is required, then

```
wc -l <file
```

can be used.

Pipelines and Filters

The standard output of one command may be connected to the standard input of another by writing the "pipe" operator, indicated by |, between

commands as in

```
ls -l | wc
```

Two or more commands connected in this way constitute a *pipeline*, and the overall effect is the same as

```
ls -l >file; wc <file
```

except that no *file* is used. Instead the two processes are connected by a pipe [see **pipe**(2)] and are run in parallel. Pipes are unidirectional, and synchronization is achieved by halting **wc**(1) when there is nothing to read and halting **ls**(1) when the pipe is full.

A *filter* is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, **grep**(1) selects from its input those lines that contain some specified string. For example,

```
ls | grep old
```

prints those lines, if any, of the output from **ls** that contain the string "old". Another useful filter is **sort**(1). For example,

```
who | sort
```

will print an alphabetically sorted list of logged in users.

A pipeline may consist of more than two commands, for example,

```
ls | grep old | wc -l
```

prints only the number of file names in the current directory containing the string "old".

File Name Generation

Many commands accept arguments which are file names. For example,

```
ls -l main.c
```

prints only information relating to the file *main.c*. The "**ls -l**" command alone prints the same information about all files in the current directory.

The **shell** provides a mechanism for generating a list of file names that match a pattern. For example,

ls −l *.c

generates as arguments to **ls**(1) all file names in the current directory that end in *.c*. The character "*" is a pattern that will match any string including the null string. In general, *patterns* are specified as follows:

*	Matches any string of characters including the null string.
?	Matches any single character.
[...]	Matches any one of the characters enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

[a−z]*

matches all names in the current directory beginning with one of the letters *a* through *z*.

The input

/usr/fred/test/?

matches all names in the directory */usr/fred/test* that consist of a single character. If no file name is found that matches the pattern then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

echo /usr/fred/*/core

finds and prints the names of all *core* files in subdirectories of */usr/fred*. [The **echo**(1) command is a standard XELOS operating system command that prints its arguments, separated by blanks.] This last feature can be expensive, requiring a scan of all subdirectories of */usr/fred*.

There is one exception to the general rules given for patterns. The character "." at the start of a file name must be explicitly matched. The input

```
echo *
```

will, therefore, echo all file names in the current directory not beginning with ".". The input

```
echo .*
```

will echo all those file names that begin with ".". This avoids inadvertent matching of the names "." and ".." which mean "the current directory" and "the parent directory", respectively. [Notice that **ls**(1) suppresses information for the files "." and "..".]

Quoting

Characters that have a special meaning to the **shell**, such as

```
< > * ? | &
```

are called *metacharacters*. A complete list of metacharacters is given in Table 5.B. Any character preceded by a \ is **quoted** and loses its special meaning, if any. The \ is elided so that

```
echo \?
```

will echo a single **?**, and

```
echo \\
```

will echo a single \. To allow long strings to be continued over more than one line, the sequence **\new line** (or RETURN) is ignored. The \ is convenient for quoting single characters. When more than one character needs quoting, the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. For example,

echo xx'****'xx

will echo

xx****xx

The quoted string may not contain a single quote but may contain new lines which are preserved. This quoting mechanism is the most simple and is recommended for casual use. A third quoting mechanism using double quotes is also available and prevents interpretation of some but not all metacharacters. Details of quoting are described under "Evaluation and Quoting" in part "Keyword Parameters".

Prompting by the Shell

When the **shell** is used from a terminal, it will issue a prompt to the terminal user indicating it is ready to read a command from the terminal. By default, this prompt is "**$** ". The prompt may be changed by entering

PS1=newprompt

This sets the prompt to be the string "newprompt". If a new line is typed and further input is needed, the **shell** will issue the prompt "> ". Sometimes this can be caused by mistyping a quote mark. If it is unexpected, then an interrupt (DEL) will return the **shell** to read another command. The other prompt (">") may be changed by entering:

PS2=more

The Shell and Login

Following the user's **login(1)**, the **shell** is called to read and execute commands typed at the terminal. If the user's login directory contains the file *.profile*, then it is assumed to contain commands and is read immediately by the **shell** before reading any commands from the terminal.

SHELL INTRODUCTION

Summary

ls Prints the names of files in the current directory.

ls >file Puts the output from **ls** into *file*.

ls | wc −l Prints the number of files in the current directory.

ls | grep old Prints those file names containing the string "old".

ls | grep old | wc −l Prints the number of files whose name contains the string "old".

cc pgm.c & Runs **cc** in the background.

SHELL PROCEDURES

The **shell** may be used to read and execute commands contained in a file. For example, the following call

```
sh file [ args ... ]
```

calls the **shell** to read commands from *file*. Such a file call is called a "command procedure" or "shell procedure". Arguments may be supplied with the call and are referred to in *file* using the *positional parameters* **$1**, **$2**, **...** . For example, if the file *wg* contains

```
who | grep $1
```

then the call

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

All XELOS operating system files have three independent attributes (often called "permissions"), *read*, *write*, and *execute* (rwx). The XELOS operating system command **chmod**(1) may be used to make a file executable. For example,

```
chmod +x wg
```

will ensure that the file *wg* has execute status (permission). Following this, the command

```
wg fred
```

is equivalent to the call

```
sh wg fred
```

This allows **shell** procedures and programs to be used interchangeably. In either case, a new process is created to execute the command.

As well as providing names for the positional parameters, the number of positional parameters in the call is available as **$#**. The name of the file being executed is available as **$0**.

A special **shell** parameter **$*** is used to substitute for all positional parameters except **$0**. A typical use of this is to provide some default arguments, as in,

```
nroff -T450 -cm $*
```

which simply prepends some arguments to those already given.

Control Flow—for

A frequent use of **shell** procedures is to loop through the arguments (**$1**, **$2**, ...) executing commands once for each argument. An example of such a procedure is *tel* that searches the file */usr/lib/telnos* that contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of *tel* is

```
for i
do
    grep $i /usr/lib/telnos
done
```

The command

```
tel fred
```

prints those lines in */usr/lib/telnos* that contain the string "fred".

The command

```
tel fred bert
```

prints those lines containing "fred" followed by those for "bert".

The **for** loop notation is recognized by the **shell** and has the general form

```
for name in w1 w2
do
    command-list
done
```

A **command-list** is a sequence of one or more simple commands separated or terminated by a new line or a semicolon. Furthermore, reserved words like **do** and **done** are only recognized following a new line or semicolon. A *name* is a **shell** variable that is set to the words *w1 w2* ... in turn each time the **command-list** following **do** is executed. If "in w1 w2 ..." is omitted, then the loop is executed once for each positional parameter; that is, **in** *$** is assumed.

Another example of the use of the **for** loop is the **create** command whose text is

```
for i do >$i; done
```

The command

```
create alpha beta
```

ensures that two empty files *alpha* and *beta* exist and are empty. The notation >*file* may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or new line) is required before **done**.

Control Flow—case

A multiple way (choice) branch is provided for by the **case** notation. For example,

```
case $# in
    1) cat >>$1 ;;
    2) cat >>$2 <$1 ;;
    *) echo ´usage: append [ from ] to´ ;;
esac
```

is an append command. (Note the use of semicolons to delimit the cases.) When called with one argument as in

```
append file
```

$# is the string "1", and the standard input is appended (copied) onto the end of *file* using the **cat**(1) command.

```
append file1 file2
```

appends the contents of *file1* onto *file2*. If the number of arguments supplied to append is other than 1 or 2, then a message is printed indicating proper usage.

The general form of the **case** command is

```
case word in
    pattern) command-list;;
    ...
esac
```

The **shell** attempts to match *word* with each *pattern* in the order in which the patterns appear. If a match is found, the associated **command-list** is executed and execution of the **case** is complete. Since * is the pattern that matches any string, it can be used for the default case.

Caution: No check is made to ensure that only one pattern matches the case argument.

The first match found defines the set of commands to be executed. In the example below, the commands following the second "*" will never be executed since the first "*" executes everything it receives.

```
case $# in
    *) ... ;;
    *) ... ;;
esac
```

Another example of the use of the **case** construction is to distinguish between different forms of an argument. The following example is a fragment of a **cc**(1) command.

```
for i
do
    case $i in
        -[ocs]) ... ;;
        -*)   echo 'unknown flag $i' ;;
        *.c)  /lib/c0 $i ... ;;
        *)    echo 'unexpected argument $i' ;;
    esac
done
```

To allow the same commands to be associated with more than one pattern, the **case** command provides for alternative patterns separated by a | . For example,

```
case $i in
    -x|-y)...
esac
```

is equivalent to

```
case $i in
    -[xy])...
esac
```

The usual quoting conventions apply so that

```
case $i in
    \?)...
```

will match the character **?**.

Here Documents

The **shell** procedure *tel* described under "A. Control Flow—for" in this section uses the file */usr/lib/telnos* to supply the data for **grep**(1). An alternative is to include this data within the **shell** procedure as a *here* document, as in,

```
for i
do
    grep $i <<!
    ...
    fred mh0123
    bert mh0789
    ...
!
done
```

In this example, the **shell** takes the lines between **<<!** and **!** as the standard input for **grep**(1). The string "!" is arbitrary. The document is being terminated by a line that consists of the string following <<.

Parameters are substituted in the document before it is made available to **grep**(1) as illustrated by the following procedure called *edg*.

```
ed $3 <<%
g/$1/s//$2/g
w
%
```

The call

```
edg string1 string2 file
```

is then equivalent to the command

```
ed file <<%
g/string1/s//string2/g
w
%
```

and changes all occurrences of "string1" in *file* to "string2". Substitution can be prevented using \ to quote the special character **$** as in

```
ed $3 <<+
1,\$s/$1/$2/g
w
+
```

[This version of *edg* is equivalent to the first except that **ed**(1) will print a **?** if there are no occurrences of the string **$1.**]

Substitution within a *here* document may be prevented entirely by quoting the terminating string, for example,

```
grep $i <<#
...
#
```

The document is presented without modification to **grep**. If parameter substitution is not required in a *here* document, this latter form is more efficient.

Shell Variables

The **shell** provides string-valued variables. Variable names begin with a letter and consist of letters, digits, and underscores. Variables may be given values by writing

```
user=fred box=123 acct=456
```

which assigns values to the variables *user, box,* and *acct*. A variable may be set to the null string by entering

```
null=
```

The value of a variable is substituted by preceding its name with **$**; for example,

```
echo $user
```

will echo *fred*.

Variables may be used interactively to provide abbreviations for frequently used strings.

For example,

```
b=/usr/fred/bin
mv file $b
```

will move the *file* from the current directory to the directory */usr/fred/bin*. A more general notation is available for parameter (or variable) substitution, as in,

```
echo ${user}
```

which is equivalent to

```
echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
tmp=/tmp/ps
ps a >${tmp}a
```

will direct the output of **ps**(1) to the file */tmp/psa,* whereas,

```
ps a >$tmpa
```

would cause the value of the variable *tmpa* to be substituted.

Except for **$?**, the following are set initially by the **shell**.

$? — The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully; otherwise, a nonzero exit status is returned. Testing the value of return codes is dealt with later under **if** and **while** commands.

$# — The number of positional parameters in decimal. Used, for example, in the **append** command to check the number of parameters.

$$ The process number of this **shell** in decimal. Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary file names. For example,

```
ps a >/tmp/ps$$
...
rm /tmp/ps$$
```

$! The process number of the last process run in the background (in decimal).

$- The current **shell** flags, such as **-x** and **-v**.

Some variables have a special meaning to the **shell** and should be avoided for general use.

$MAIL When used interactively, the **shell** looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at, the **shell** prints the message "you have mail" before prompting for the next command. This variable is typically set in the file *.profile* in the user's login directory. For example:

```
MAIL=/usr/mail/fred
```

$HOME The default argument for the **cd**(1) command. The current directory is used to resolve file name references that do not begin with a / and is changed using the **cd** command.

For example,

```
cd /usr/fred/bin
```

makes the current directory */usr/fred/bin*. Then

```
cat wn
```

will print on the terminal the file *wn* in this directory. The command **cd**(1) with no argument is equivalent to

cd $HOME

This variable is also typically set in the user's login profile.

$PATH A list of directories containing commands (the *search path*). Each time a command is executed by the **shell**, a list of directories is searched for an executable file. If *$PATH* is not set, the current directory, */bin*, and */usr/bin* are searched by default. Otherwise, *$PATH* consists of directory names separated by **:**. For example,

PATH=:/usr/fred/bin:/bin:/usr/bin

specifies that the current directory (the null string before the first **:**), */usr/fred/bin*, */bin*, and */usr/bin* are to be searched in that order. In this way, individual users can have their own 'private' commands that are accessible independently of the current directory. If the command name contains a **/**, this directory search is not used; a single attempt is made to execute the command.

$PS1 The primary **shell** prompt string, by default, "**$** ".

$PS2 The **shell** prompt when further input is needed, by default, "> ".

$IFS The set of characters used by *blank interpretation*. (See "D. Evaluation and Quoting" in part "Keyword Parameters".)

Test Command

The **test** command is intended for use by **shell** programs. For example,

test −f file

returns zero exit status if *file* exists and nonzero exit status otherwise. In general, **test** evaluates a predicate and returns the result as its exit status. Some of the more frequently used **test** arguments are given below [see **test**(1) for a complete specification].

test s	true if the argument *s* is not the null string
test −f file	true if *file* exists
test −r file	true if *file* is readable
test −w file	true if *file* is writable
test −d file	true if *file* is a directory

Control Flow—while

The actions of the **for** loop and the **case** branch are determined by data available to the **shell**. A **while** or **until** loop and an **if then else** branch are also provided, whose actions are determined by the exit status returned by commands.

A **while** loop has the general form

```
while command-list1
do
    command-list2
done
```

The value tested by the **while** command is the exit status of the last simple command following **while**. Each time around the loop *command-list1* is executed; if a zero exit status is returned, then *command-list2* is executed; otherwise, the loop terminates. For example,

```
while test $1
do
    ...
    shift
done
```

is equivalent to

```
for i
do
    ...
done
```

The **shift** command is a **shell** command that renames the positional parameters **$2, $3, ...** as **$1, $2, ...** and loses **$1.**

Another kind of use for the **while/until** loop is to wait until some external event occurs and then run some commands. In an **until** loop, the termination condition is reversed. For example,

```
until test -f file
do
    sleep 300
done
commands
```

will loop until *file* exists. Each time around the loop, it waits for 5 minutes (300 seconds) before trying again. (Presumably, another process will eventually create the file.)

Control Flow—if

Also available is a general conditional branch of the form,

```
if command-list
then
  command-list
else
  command-list
fi
```

that tests the value returned by the last simple command following **if**.

The **if** command may be used in conjunction with the **test** command to test for the existence of a file as in

```
if test -f file
then
        process file
else
        do something else
fi
```

An example of the use of **if, case,** and **for** constructions is given in "I. The Man Command" in part "Shell Procedures".

A multiple test **if** command of the form

```
if ...
then
        ...
else
        if ...
        then
                ...
        else
                if ...
                ...
                fi
        fi
fi
```

may be written using an extension of the **if** notation as,

```
if ...
then
        ...
elif ...
then
        ...
elif ...
...
fi
```

The **touch** command changes the "last modified" time for a list of files. The command may be used in conjunction with **make**(1) to force recompilation of a list of files.

The following example is the **touch** command:

```
flag=
for i
do
    case $i in
        -c)   flag=N ;;
        *)    if test -f $i
              then
                  ln $i junk$$
                  rm junk$$
              elif test $flag
              then
                  echo file \'$i\' does not exist
              else
                  >$i
              fi ;;
    esac
done
```

The **−c** flag is used in this command to force subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The shell variable *flag* is set to some non-null string if the **−c** argument is encountered. The commands

```
ln ...; rm ...
```

make a link to the file and then remove it.

The sequence

```
if command1
then
      command2
fi
```

may be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

executes **command2** only if **command1** fails. In each case, the value returned is that of the last simple command executed.

Command Grouping

Commands may be grouped in two ways,

{ *command-list* ; }

and

(*command-list*)

The first form, *command-list*, is simply executed. The second form executes *command-list* as a separate process. For example,

```
(cd x; rm junk )
```

executes *rm junk* in the directory *x* without changing the current directory of the invoking shell.

The commands

```
cd x; rm junk
```

have the same effect but leave the invoking shell in the directory *x*.

Debugging Shell Procedures

The **shell** provides two tracing mechanisms to help when debugging shell procedures. The first is invoked within the procedure as

```
set −v
```

(*v* for verbose) and causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by entering

```
sh −v proc ...
```

where *proc* is the name of the **shell** procedure. This flag may be used in conjunction with the **−n** flag which prevents execution of subsequent commands. (Note that typing "**set −n**" at a terminal will render the terminal useless until an end of file is typed.)

The command

```
set −x
```

will produce an execution trace with flag **−x**. Following parameter substitution, each command is printed as it is executed. (Try the above at the terminal to see the effect it has.) Both flags may be turned off by typing

```
set −
```

and the current setting of the **shell** flags is available as **$−** .

The "man" Command

The following discussion of the man command assumes the existence of the document preparation features available as an option on the XELOS system.

The following is the **man** command which is used to print sections of the *XELOS User Reference Manual*. It is called by entering

```
man sh
man −t ed
man 2 fork
```

In the first call, the manual section for **sh** is printed. Since no section is specified, section 1 is used. The second call will typeset (**−t** option) the manual section for **ed**. The last call prints the **fork** manual page from section 2 of the manual.

KEYWORD PARAMETERS

Shell variables may be given values by assignment or when a **shell** procedure is invoked. An argument to a **shell** procedure of the form *name=value* that precedes the command name causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking **shell** is not affected. For example,

```
user=fred command
```

will execute **command** with *user* set to *fred*. The **−k** flag causes arguments of the form *name=value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain, they are available as positional parameters **$1, $2, ...** .

The **set** command may also be used to set positional parameters from within a procedure.

For example,

```
set − *
```

will set **$1** to the first file name in the current directory, **$2** to the next, etc. Note that the first argument, −, ensures correct treatment when the first file name begins with a − .

Parameter Transmission

When a **shell** procedure is invoked, both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a **shell** procedure by specifying in advance that such parameters are to be exported. For example,

```
export user box
```

marks the variables *user* and *box* for export. When a **shell** procedure is invoked, copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking **shell**. It is generally true of a **shell** procedure that it may not modify the state of its caller without an explicit request on the part of the caller. (Shared file descriptors are an exception to this rule.)

Names whose value is intended to remain constant may be declared *readonly*. The form of this command is the same as that of the **export** command,

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

Parameter Substitution

If a **shell** parameter is not set, then the null string is substituted for it. For example, if the variable *d* is not set,

```
echo $d
```

or

```
echo ${d}
```

will echo nothing. A default string may be given as in

```
echo ${d-.}
```

which will echo the value of the variable *d* if it is set and "." otherwise.

The default string is evaluated using the usual quoting conventions so that

```
echo ${d- '*'}
```

will echo * if the variable *d* is not set. Similarly,

```
echo ${d-$1}
```

will echo the value of *d* if it is set and the value (if any) of **$1** otherwise. A variable may be assigned a default value using the notation

```
echo ${d=.}
```

which substitutes the same string as

```
echo ${d-.}
```

and if *d* were not previously set, it will be set to the string ".". (The notation ${...=...} is not available for positional parameters.)

If there is no sensible default, the notation

```
echo ${d?message}
```

will echo the value of the variable *d* if it has one; otherwise, *message* is printed by the **shell** and execution of the **shell** procedure is abandoned. If *message* is absent, a standard message is printed. A **shell** procedure that requires some parameters to be set might start as follows:

```
: ${user?} ${acct?} ${bin?}
...
```

Colon (:) is a command built into the **shell** and does nothing once its arguments have been evaluated. If any of the variables *user, acct,* or *bin* are not set, the **shell** will abandon execution of the procedure.

Command Substitution

The standard output from a command can be substituted in a similar way to parameters. The command **pwd(1)** prints on its standard output the name of the current directory. For example, if the current directory is

/usr/fred/bin, the command

```
d='pwd'
```

is equivalent to

```
d=/usr/fred/bin
```

The entire string between single quotes ('...') is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a ' must be escaped using a \.

For example,

```
ls 'echo "$1"'
```

is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including *here* documents), and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within **shell** procedures. An example of such a command is **basename**, which removes a specified suffix from a string. For example,

```
basename main.c .c
```

will print the string "main". Its use is illustrated by the following fragment from a **cc**(1) command.

```
case $A in
        ...
        *.c)      B='basename $A .c'
        ...
esac
```

that sets **B** to the part of **$A** with the suffix **.c** stripped.

Here are some composite examples.

- for i in 'ls −t'; do ...

The variable *i* is set
to the names of files in time order,
most recent first.

- set 'date'; echo $6 $2 $3, $4

will print, e.g.,
1977 Nov 1, 23:59:59

Evaluation and Quoting

The **shell** is a macro processor that provides parameter substitution, command substitution, and file name generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in Table 5.A. Before a command is executed, the following substitutions occur:

1. Parameter substitution, e.g., **$user**
2. Command substitution, e.g., **'pwd'**

   Only one evaluation occurs so that if, for example, the value of the variable *X* is the string "$y" then

   echo $X

   will echo "$y".
3. Blank interpretation

   Following the above substitutions, the resulting characters are broken into nonblank words (*blank interpretation*). For this purpose, 'blanks' are the characters of the string "*$IFS*". By default, this string consists of blank, tab, and newline. The null string is not regarded as a word unless it is quoted. For example,

   echo ´´

will pass on the null string as the first argument to **echo**, whereas

echo $null

will call **echo** with no arguments if the variable *null* is not set or set to the null string.

4. File name generation

   Each word is then scanned for the file pattern characters *, ?, and [...]; and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

The evaluations just described also occur in the list of words associated with a **for** loop. Only substitution occurs in the *word* used for a case branch.

As well as the quoting mechanisms described earlier using \ and ´...´, a third quoting mechanism is provided using double quotes. Within double quotes, parameter and command substitution occurs; but file name generation and the interpretation of blanks does not.

The following characters have a special meaning within double quotes and may be quoted using \.

$	parameter substitution
‘	command substitution
"	ends the quoted string
\	quotes the special characters $ ‘ " \

For example,

echo "$x"

will pass the value of the variable *x* as a single argument to **echo.** Similarly,

echo "$*"

will pass the positional parameters as a single argument and is equivalent to

echo "$1 $2 ..."

The notation $@ is the same as $* except when it is quoted. Inputting

```
echo "$@"
```

will pass the positional parameters, unevaluated, to **echo** and is equivalent to

```
echo "$1" "$2" ...
```

The following illustration gives, for each quoting mechanism, the shell metacharacters that are evaluated.

metacharacter

	\	$	*	‘	"	´
´	n	n	n	n	n	t
‘	y	n	n	t	n	n
"	y	y	n	y	t	n

t = terminator
y = interpreted
n = not interpreted

In cases where more than one evaluation of a string is required, the built-in command **eval** may be used. For example, if the variable *X* has the value "$y" and if *y* has the value "pqr", then

```
eval echo $X
```

will echo the string "pqr".

In general, the **eval** command evaluates its arguments (as do all commands) and treats the result as input to the **shell**. The input is read and the resulting command(s) executed. For example,

```
wg=´eval who | grep´
$wg fred
```

is equivalent to

```
who | grep fred
```

In this example, **eval** is required since there is no interpretation of metacharacters, such as |, following substitution.

Error Handling

The treatment of errors detected by the **shell** depends on the type of error and on whether the **shell** is being used interactively. An interactive **shell** is one whose input and output are connected to a terminal [as determined by **gtty**(2)]. A **shell** invoked with the **−i** flag is also interactive.

Execution of a command (see also "G. Command Execution") may fail for any of the following reasons:

- Input/output (I/O) redirection may fail. For example, if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a "bus error" or "memory fault" signal.
- The command terminates normally but returns a nonzero exit status.

In all of these cases, the **shell** will go on to execute the next command. Except for the last case, an error message will be printed by the **shell**. All remaining errors cause the **shell** to exit from a command procedure. An interactive **shell** will return to read another command from the terminal. Such errors include the following:

- Syntax errors, e.g., if ...then... done
- A signal such as interrupt. The **shell** waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands such as **cd**(1).

The **shell** flag **−e** causes the **shell** to terminate if any error is detected. The following is a list of the XELOS operating system signals:

1	hangup
2	interrupt
3*	quit
4*	illegal instruction
5*	trace trap
6*	IOT instruction
7*	EMT instruction
8*	floating point exception
9	Kill (cannot be caught or ignored)
10*	bus error
11*	segmentation violation
12*	bad argument to system call
13	write on a pipe with no one to read it
14	alarm clock
15	software termination [from **kill**(1)]

The XELOS operating system signals marked with an asterisk "*" as shown in the list produce a core dump if not caught. However, the shell itself ignores quit which is the only external signal that can cause a dump. The signals in this list of potential interest to **shell** programs are 1, 2, 3, 14, and 15.

Fault Handling

Shell procedures normally terminate when an interrupt is received from the terminal. The **trap** command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap ´rm /tmp/ps$$; exit´ 2
```

sets a trap for signal 2 (terminal interrupt); and if this signal is received, it will execute the following commands:

```
rm /tmp/ps$$; exit
```

The **exit** is another built-in command that terminates execution of a shell procedure. The **exit** is required; otherwise, after the trap has been taken, the **shell** will resume executing the procedure at the place where it was interrupted.

XELOS operating system signals can be handled in one of three ways.

1. They can be ignored, in which case the signal is never sent to the process.
2. They can be caught, in which case the process must decide what action to take when the signal is received.
3. They can be left to cause termination of the process without it having to take any further action.

If a signal is being ignored on entry to the **shell** procedure, for example, by invoking it in the background (see "G. Command Execution"), **trap** commands (and the signal) are ignored.

The use of **trap** is illustrated by the following modified version of the **touch** command:

```
flag=
trap ´rm −f junk$$; exit´ 1 2 3 15
for i
do
    case $i in
    −c)  flag=N ;;
    *)  if test −f $i
        then
                ln $i junk$$; rm junk$$
        elif test $flag
        then
                echo file \´$i\´ does not exist
        else
                >$i
        fi ;;
    esac
done
```

The cleanup action is to remove the file *junk$$*. The **trap** command appears before the creation of the temporary file; otherwise, it would be possible for the process to die without removing the file.

Since there is no signal 0 in the XELOS operating system, it is used by the shell to indicate the commands to be executed on exit from the shell procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to trap. The following:

```
trap ´´ 1 2 3 15
```

is a fragment taken from the **nohup(1)** command which causes the XELOS operating system HANGUP, INTERRUPT, QUIT, and SOFTWARE TERMINATION signals to be ignored both by the procedure and by invoked commands. Traps may be reset by entering

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing

```
trap
```

The **scan** procedure is an example of the use of **trap** where there is no exit in the trap command. The **scan** takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when **scan** is waiting for input. The **scan** procedure follows:

```
d=`pwd`
for i in *
do
        if test -d $d/$i
        then
              cd $d/$i
              while echo "$i:" && trap exit 2 && read x
              do
                      trap : 2
                      eval $x
              done
        fi
done
```

The **read x** is a built-in command that reads one line from the standard input and places the result in the variable *x*. It returns a nonzero exit status if either an end-of-file is read or an interrupt is received.

Command Execution

To run a command (other than a built-in), the **shell** first creates a new process using the system call **fork(2)**. The execution environment for the command includes input, output, and the states of signals and is established in the child process before the command is executed. The built-in command **exec** is used in rare cases when no fork is required and simply replaces the **shell** with a new command. For example, a simple version of the **nohup** command looks like

```
trap '' 1 2 3 15
exec $*
```

The **trap** turns off the signals specified so that they are ignored by subsequently created commands, and **exec** replaces the **shell** by the command specified.

Most forms of I/O redirection have already been described. In the following, *word* is only subject to parameter and command substitution. No file name generation or blank interpretation takes place so that, for example,

```
echo ... >*.c
```

will write its output into a file whose name is **.c*. I/O specifications are evaluated left-to-right as they appear in the command. Some I/O specifications are as follows:

> *word*	The standard output (file descriptor 1) is sent to the file *word* which is created if it does not already exist.
>> *word*	The standard output is sent to file *word*. If the file exists, then output is appended (by seeking to the end); otherwise, the file is created.
< *word*	The standard input (file parameter 0) is taken from the file *word*.
<< *word*	The standard input is taken from the lines of **shell** input that follow up to but not including a line consisting only of *word*. If *word* is quoted, no interpretation of the document occurs. If *word* is not quoted, parameter and command substitution occur and \ is used to quote the characters \, $, ‘, and the first character of *word*. In the latter case, \newline is ignored (e.g., quoted strings).
>& *digit*	The file descriptor *digit* is duplicated using the system call **dup**(2), and the result is used as the standard output.

<& *digit* — The standard input is duplicated from file descriptor *digit*.

<&- — The standard input is closed.

>&- — The standard output is closed.

Any of the above may be preceded by a digit, in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
... 2>file
```

runs a command with message output (file descriptor 2) directed to *file*. Another example,

```
... 2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking, file descriptor 2 is created by duplicating file descriptor 1; but the effect is usually to merge the two streams.)

The environment for a command run in the background such as

```
list *.c | lpr &
```

is modified in two ways. First, the default standard input for such a command is the empty file */dev/null*. This prevents two processes (the **shell** and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

```
ed file &
```

would allow both the editor and the **shell** to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason, the XELOS operating system convention for a signal is that if it is set to 1 (ignored) then it is never changed even for a short time. Note that the **shell**

command **trap** has no effect for an ignored signal.

Invoking the Shell

The following flags are interpreted by the **shell** when it is invoked. If the first character of argument zero is a minus, commands are read from the file *.profile*.

−c *string*	If the **−c** flag is present, then commands are read from *string*.
−s	If the **−s** flag is present or if no arguments remain, commands are read from the standard input. Shell output is written to file descriptor 2.
−i	If the **−i** flag is present or if the **shell** input and output are attached to a terminal [as told by **getty**(8)], this **shell** is *interactive*. In this case, TERMINATE is ignored (so that **kill 0** does not kill an interactive **shell**, and INTERRUPT is caught and ignored (so that **wait** is interruptible). In all cases, QUIT is ignored by the **shell**.

TABLE 5.A

GRAMMAR

item	*word* *input-output* *name = value*
simple-command:	*item* *simple-command item*
command:	*simple-command*

(*command-list*)
{ *command-list* }
for *name* **do** *command-list* **done**
for *name* **in** *word* ... **do** *command-list* **done**
while *command-list* **do** *command-list* **done**
until *command-list* **do** *command-list* **done**
case *word* **in** *case-part* ... **esac**
if *command-list* **then** *command-list else-part* **fi**

pipeline:
command
pipeline | *command*

andor:
pipeline
andor && *pipeline*
andor || *pipeline*

command-list:
andor
command-list ;
command-list &
command-list ; *andor*
command-list & *andor*

input-output:
> *word*
< *word*
>> *word*
<< *word*

file
word
& *digit*
& −

case-part:
pattern) *command-list* ;;

pattern:
word
pattern | *word*

else-part:
elif *command-list* **then** *command-list else-part*
else *command-list*

empty:
empty

word:
sequence of nonblank characters

name	sequence of letters, digits, or underscores starting with a letter
digit:	**0 1 2 3 4 5 6 7 8 9**

TABLE 5.B

METACHARACTERS AND RESERVED WORDS

(a) *syntactic:*

|	pipe symbol
&&	'andf' symbol
||	'orf' symbol
;	command separator
;;	case delimiter
&	background commands
( )	command grouping
<	input redirection
<<	input from a here document
>	output creation
>>	output append

(b) *patterns:*

*	match any character(s) including none
?	match any single character
[...]	match any of the enclosed characters

(c) *substitution:*

${...}	substitute shell variable
\`...\`	substitute command output

(d) *quoting:*

\	quote the next character
'...'	quote the enclosed characters except for the '

"..." quote the enclosed characters except for the $, ‘ ,\, and "

(e) *reserved words:*

if then else elif fi
case in esac
for while until do done
{ } [ ] test

Chapter 6

GLOSSARY

The following list defines terms and acronyms used in this volume which may not be familiar to the user.

argument – Words following the command on a command line that provide information necessary to execute a program. Command arguments are very often file names.

ASCII – American Standard Code for Information Interchange.

background – A mode of program execution when the shell does not wait for the command to terminate before prompting for another command.

C language – A general-purpose, low level programming language used to write programs (such as numerical, text-processing, and data base) and operating systems (such as the XELOS operating system).

command – The first word of a command line. It is the name of an executable file that is a compiled program.

command line – A sequence of nonblank arguments separated by blanks or tabs typed in by a user. The first argument usually specifies the name of a command.

command list – A sequence of one or more simple commands separated or terminated by a new line or a semicolon.

command procedure – A command procedure is an executable file that is not a compiled program. It is a call to the **shell** to read and execute commands contained in a file. A sequence of commands may thus be preserved for repeated use by saving it in a file which can also be called a shell procedure, a command file, or a runcom according to local preference.

command substitution – When the **shell** reads a command line, any command or commands enclosed between grave accents ('...') are executed first and the output from these commands replace the whole expression ('...').

current working directory – The current point of reference for accessing data within the file system.

directory – A type of file that is used to group and organize files and other directories.

EOF – The end of file character is the same as an ASCII EOT character. See EOT.

EOT – The end of text character is generated by holding down the "CONTROL" key and pressing the lower-case "d" key once. The EOT is used to terminate the **shell** which usually logs a user off the system.

erase character – The character which is used to delete the previous character on the current line. To turn off the special meaning of the erase character, it must be preceded with a "\". By default, the erase character is #. See **stty**(1) to change the default character.

file – An organized collection of information containing data, programs, or both which allows users to store, retrieve, and modify information. A simple file name is a sequence of characters other than a slash (/).

filter – A command that reads its standard input, transforms it in some way, and prints the result as output.

foreground – A mode of program execution when the shell waits for the command to terminate before prompting for another command.

full pathname – The pathname of a specific file starting from the **root** directory.

group identification number (gid) – A unique number assigned to one or more logins that is used to identify groups of related users.

here documents – A command procedure that has the form **command << eofstring** which causes the **shell** to read subsequent lines as standard input to the command until a line is read consisting of only the **eofstring**. Any arbitrary string can be used for the **eofstring**.

HOME – Another name for the login directory.

in-line input documents – See here documents.

keyword parameters – An argument to a command procedure of the form **name=value command arg1 arg2 . . .** here **name** is called the keyword parameter. This allows **shell** variables to be assigned values when a **shell** procedure is called. The value of **name** in the invoking **shell** is not affected, but the value is assigned to **name** before execution of the procedure. The arguments (arg1 arg2 . . .) are available as positional parameters ($1 $2 . . .).

kill character – The character which is used to delete all the characters typed before it on the current line. To turn off the special meaning of the kill character, it must be preceded with a "\". By default, the kill character is @. The default character can be changed via **stty(1)**.

login – A means by which a user can gain access to the XELOS operating system.

login name – A unique string of letters and numbers used to identify a login.

log off – A procedure to disconnect the user from the XELOS operating system.

memorandum macros – The standard, general-purpose package of text formatting macros used in conjuction with nroff and troff to produce

many common types of documents.

metacharacters – Characters that have a special meaning to the **shell**, such as <, >, *, ?, |, &, $, ;, (,), \, ", ‘, ´, [,], etc.

mode – An absolute mode is an octal number used in conjunction with **chmod**(1) to change permissions of files.

nroff – A text formatting program for driving typewriter-like terminals and printers to produce a screen copy or a hardcopy.

parent directory – The directory immediately above another directory. A ".." is the shorthand name for the parent directory. To make the parent directory of your current working directory your new current directory enter the "**cd ..**" command.

partial pathname – The pathname between the current working directory and a specific file.

password – A string of up to 13 characters chosen from a 64 character alphabet (., \, 0-9, A-Z, a-z).

pathname – A sequence of directory names separated by the / character and ending with the name of a file. The pathname defines the connection path between some directory and a file.

pipe – A simple way to connect the output of one program to the input of another program, so that each program will run as a sequence of processes.

pipeline – A series of filters separated by the character |. The output of each filter becomes the input of the next filter in the line. The last filter in the line will write to its standard output.

positional parameters – Arguments supplied with a command procedure that are placed into variable names $1, $2, . . . when the command

procedure is invoked by the **shell**. The name of the file being executed is positional parameter $0.

primary prompt – A notification (by default "$ ") to the user that the XELOS operating system **shell** is ready to accept another request.

process – A program that is in some state of execution. The execution of a computer environment including contents of memory, register values, name of the current directory, status of open files, information recorded at login time, and various other items.

program – Software that can be executed by a user.

secondary prompt – A notification (by default "> ") to the user that the command typed in response to the primary prompt is incomplete.

shell – A XELOS system user program written in C language that handles the communication between the system and users. The shell accepts commands and causes the appropriate program to be executed.

shell procedure – See command procedure.

standard input – The standard input of a command is sent to an open file which is normally connected to the keyboard. An argument to the **shell** of the form "< file" opens the specified file as the standard input thus redirecting input to come from the file named instead of the keyboard.

standard output – Output produced by most commands is sent to an open file which is normally connected to the printer or screen. This output may be redirected by an argument to the **shell** of the form "> file" which opens the specified file as the standard output.

text editor – An interactive program (**ed**) for creating and modifying text, using commands provided by a user at a terminal.

troff – A text formatting program for driving a phototypesetter to

produce high quality printed text.

user-defined variables – A user variable can be defined using an assignment statement of the form **name=value** where **name** must begin with a letter or underscore and may then consist of any sequence of letters, digits, or underscores up to 512 characters. The **name** is the variable. Positional parameters cannot be in the name.

user identification number (uid) – A unique number assigned to each login that is used to identify users and the owner of information stored on the system.

variables – A variable is a name representing a string value. Variables which are normally set only on a command line are called parameters (positional parameters and keyword parameters). Other variables are simply names to which the user (user-defined variables) or the **shell** itself may assign string values.