TEK USERS MANUAL

Part No. 070-4730-00
Product Group 61

8560 Series
Multi-User Software
Development
Unit

TNIX

System Users Manual

TNIX Version 2

Please check for change information at the rear of this manual

Printing JUN 1983
Revised AUG 1983

Tektronix
COMMITTED TO EXCELLENCE

LIMITED RIGHTS LEGEND

Software License No.

Contractor: Tektronix, Inc.
Explanation of Limited Rights Data Identification Method
Used: Entire document subject to limited rights.

Those portions of this technical data indicated as limited rights data shall not, without the written permission of the above Tektronix, be either (a) used, released or disclosed in whole or in part outside the Customer, (b) used in whole or in part by the Customer for manufacture or, in the case of computer software documentation, for preparing the same or similar computer software, or (c) used by a party other than the Customer, except for: (i) emergency repair or overhaul work only, by or for the Customer, where the item or process concerned is not otherwise reasonably available to enable timely performance of the work, provided that the release or disclosure hereof outside the Customer shall be made subject to a prohibition against further use, release or disclosure; or (ii) release to a foreign government, as the interest of the United States may require, only for information or evaluation within such government or for emergency repair or overhaul work by or for such government under the conditions of (i) above. This legend, together with the indications of the portions of this data which are subject to such limitations shall be included on any reproduction hereof which includes any part of the portions subject to such limitations.

RESTRICTED RIGHTS IN SOFTWARE

The software described in this document is licensed software and subject to **restricted rights**. The software may be used with the computer for which or with which it was acquired. The software may be used with a backup computer if the computer for which or with which it was acquired is inoperative. The software may be copied for archive or backup purposes. The software may be modified or combined with other software, subject to the provision that those portions of the derivative software incorporating restricted rights software are subject to the same restricted rights.

Products of Tektronix, Inc. and its subsidiaries are covered by U.S. and foreign patents and/or pending patents.

Printed in U.S.A. Specification and price change privileges are reserved.

GUIDE TO DOCUMENTATION

This page shows the manuals you are most likely to use with your 8560 Series system. We recommend that you acquaint yourself with each of these manuals. (You probably won't read any manual all the way through, but we do suggest that you acquire a general idea of which information is contained in which manual.) Section 1 of each manual contains pointers to the rest of the information in the manual.

This manual explains TNIX, the operating system of your 8560 Series system, and describes standard 8560 Series features.

8560 Series MUSDU System Users Manual

This manual tells how to unpack and install the 8560 Series system. It also explains the operations to be done by the system manager—the person responsible for connecting 8540s and 8550s and maintaining accounts, software, and other aspects of a multi-user system.

8560 Series MUSDU System Manager's Operation and Installation Guide

The TNIX operating system contains an online "manual page" of information about commands. You can show the information on the terminal screen, or print it on your printer.

8560 Series Online Documentation

This optional accessory manual contains printed versions of the online manual pages for standard TNIX commands. You may want to order this manual if you find yourself printing many of the online manual pages.

8560 Series MUSDU System Reference Manual

4730-1

In addition to the above documentation, you may also be using manuals for other 8500 Series instruments (8540s and 8550s) or software products. The *Learning Guide* of the *8560 Series System Users Manual* contains a list of user manuals for many products used with 8560 Series systems.

ABOUT THIS MANUAL

This manual is your guide to using the 8560 and 8561 Multi-User Software Development Units. (Unless noted otherwise, all references to the 8560 refer to the 8561 as well.) In this manual, you'll find an overview of the 8560 system, as well as detailed information on the TNIX operating system and all standard 8560 features.

This manual is one of several sources of information about your 8560. A companion volume, the *8560 Series System Manager's Guide*, shows how to install and check out your 8560 and how to configure workstations and peripherals around an 8560. If your 8560 system is not installed, refer to the *System Manager's Guide* before you go any further. The *System Manager's Guide* also shows how to perform system maintenance activities, such as creating user accounts, installing software, and verifying disk integrity.

An optional *8560 Series System Reference Manual* contains a detailed description of each TNIX command. In addition, you can obtain online information about TNIX commands and other topics.

This *System Users Manual* is organized as follows:

Section 1. Learning Guide. Describes the 8560 system and helps you get started using the 8560, the Keyshell interface, and the TNIX operating system. Also describes online help tools.

Section 2. TNIX Operating System. Describes the TNIX file system and command language.

Section 3. Operating Procedures. Describes tasks frequently performed on the 8560 and the commands that perform those tasks.

Section 4. Shell Programming. Shows how to use the 8560's shell programming language.

Section 5. TNIX Editor. Describes the standard TNIX editor, **ed.**

Section 6. Maintaining Files. Explains how to use the **make** utility program to keep program modules and other files up-to-date.

Section 7. Communication with 8540s and 8550s. Discusses communication between your 8560 and a TEKTRONIX 8540 Integration Unit or 8550 Microcomputer Development Lab.

Section 8. Keyshell. Describes technical information about Keyshell, a program that simplifies the task of entering TNIX commands.

Section 9. Commands. Contains a brief summary of all standard TNIX commands.

Section 10. Error Messages. Explains error messages that may be issued by TNIX commands.

Section 11. Glossary.

Section 12. Index.

CONTENTS

Section 1
LEARNING GUIDE

ILLUSTRATIONS

TABLES

Section 1

LEARNING GUIDE

INTRODUCTION

This Learning Guide provides an overview of the TEKTRONIX 8560 Series Multi-User Software Development Unit, and helps you get started accomplishing your tasks on the 8560.

NOTE

Unless noted otherwise, all references in this manual to the 8560 also refer to the 8561. In addition, the term "8560 Series development system" refers to both the 8560 and 8561.

NOTE

This manual assumes that your 8560 has been unpacked, installed, and verified, and that your system terminal, other peripherals, and workstations have been configured to communicate with the 8560. If any part of your system is not ready for use, refer to your ***8560 Series System Manager's Guide*** *for instructions. Installation should be performed only by a qualified service technician.*

This section is organized into the following topics:

- **The 8560 System.** Explains the role of the 8560 in developing microprocessor-based products, and describes the minimum 8560 system and available options.
- **Getting Started.** Includes a tutorial and general information that help you quickly become proficient at using the 8560 and its Keyshell interface.
- **For Continued Learning.** Helps you decide where to go for further information on the 8560.

The next few pages provide an overview of the 8560 and describe options available for use with your 8560. If you'd rather start using the 8560 immediately, skip ahead to the discussion "Getting Started" later in this section.

THE 8560 SYSTEM

The development of a microprocessor-based product proceeds through three principal phases. During **hardware development**, a hardware prototype of the product is designed and constructed. During **software development**, software engineers design and create the program that will execute on the microprocessor that controls the product. Finally, during **hardware/software integration**, the software is executed in the prototype hardware, and the hardware and software are modified to correct any problems.

Figure 1-1 illustrates this process, and shows how you can use the 8560 and a TEKTRONIX workstation (such as the 8540 Integration Unit) throughout the software development and hardware/software integration phases of the design cycle. The following paragraphs provide an overview of what the 8560 provides during these two phases.

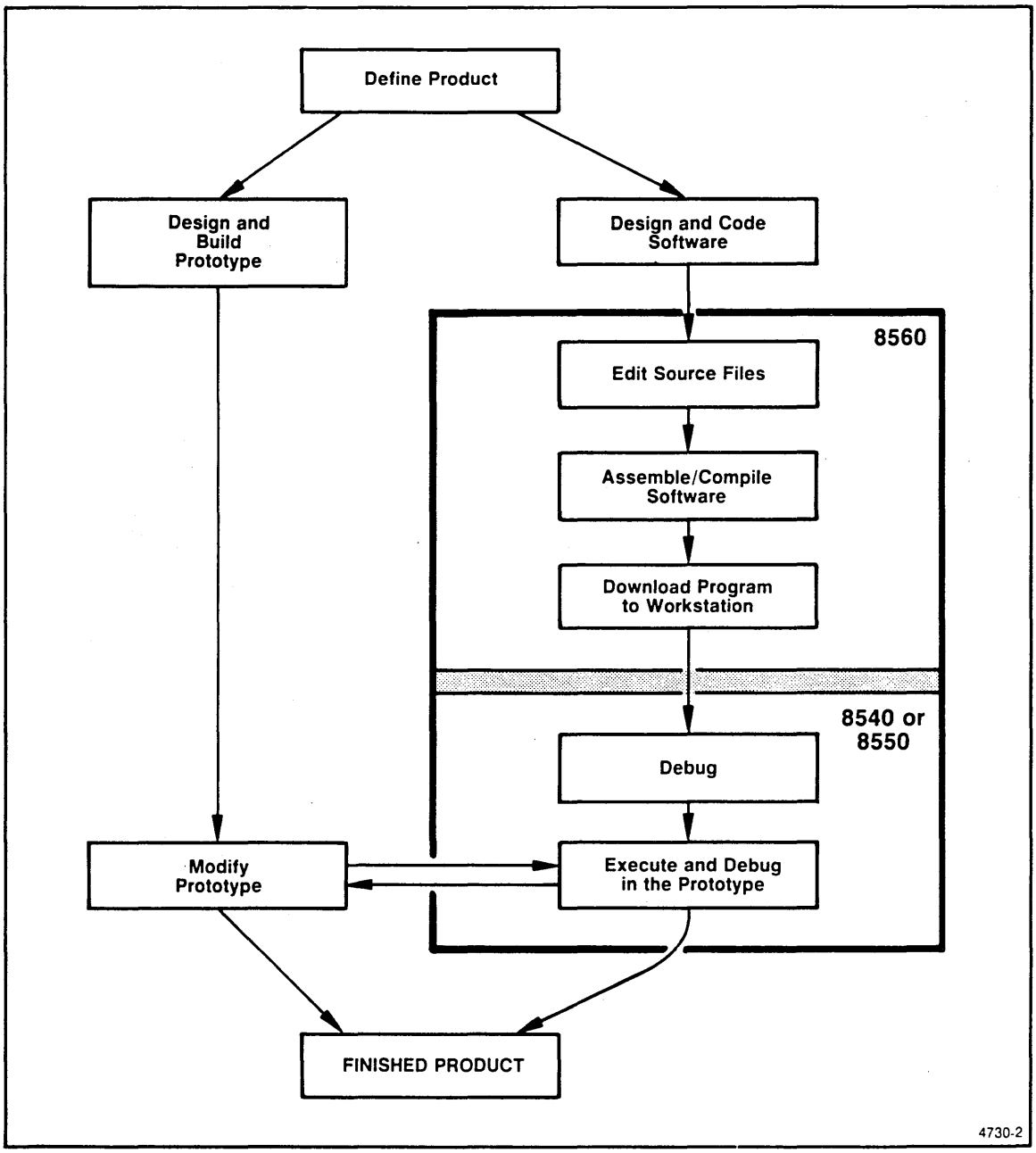


Fig. 1-1. Role of 8560 system in product design.

The 8560, in conjunction with a workstation such as the 8540 or 8550, can be used for tasks shown within the heavy black box.

During Software Development

The 8560 offers mass storage and an extensive array of modular software development tools, text processors, and general utilities. The 8560's TNIX operating system enables you to combine these tools to accomplish whatever task is at hand.[1]

The following paragraphs list some of the capabilities of the TNIX operating system:

- TNIX encourages a team approach to software development. The shared file system provides a common database, but also guarantees individual and group file protection.
- A hierarchical file system makes it easy to organize and share files.
- Team members can communicate over the system, both by electronic mail and by direct communication from one terminal to another. In addition, the system manager can post a "message of the day" and can broadcast messages to all current users.
- Program maintenance facilities automatically update complex groups of interdependent program modules. You save time because you don't have to keep track of interdependent files and individually compile or assemble them.
- It's easy to create custom tools and make them available to all users on the system or on your design team.
- The multi-tasking environment allows several commands to execute at once and provides full line-printer spooling.
- Command input and output can come from files or other commands, as well as from the terminal. This reduces the need for temporary files, and provides flexibility and convenience in your interactions with the system.
- Input and output operations are simplified because all devices are treated as "special files".
- Full type-ahead lets you enter several commands without waiting for the first command to execute.

These features are described in the remaining sections of this manual and in the *8560 Series System Manager's Guide*.

During Hardware/Software Integration

In combination with one or more workstations, the 8560 forms a complete development system. In order to execute your program, the 8560 must communicate with a workstation such as the 8540 Integration Unit or the 8550 Microcomputer Development Lab. During program execution, the workstation emulates the functions of the microprocessor for which the program was written. This allows you to monitor your program in real time as it interacts with prototype hardware, and to discover any errors that may remain in the software or hardware.

During the integration phase, you will probably use tools such as the Trigger Trace Analyzer and Digital Design Lab to monitor and analyze program performance. The 8560 command set includes data reduction tools that aid in analyzing the data generated by these debugging and analysis tools.

[1] **TNIX is a trademark of Tektronix, Inc. TNIX is derived from Western Electric Version 7 of the UNIX operating system. UNIX is a trademark of Bell Laboratories.**

The 8560 can support up to four terminals or workstations (eight if you have the Eight-User Upgrade option). The 8561 supports one or two users, and can be upgraded to support four or eight users. Each terminal or workstation connects to one of the HSI I/O ports on the rear panel of the 8560. This arrangement gives you access both to TNIX and to the workstation's operating system—regardless of whether the terminal is attached to the workstation or directly to the 8560. Figure 1-2 shows one possible multi-workstation configuration for the 8560.

For information on how to configure the 8560 to communicate with workstations, refer to Section 7 of this manual, *Communication with 8540s and 8550s*, and to the *8560 Series System Manager's Guide*.

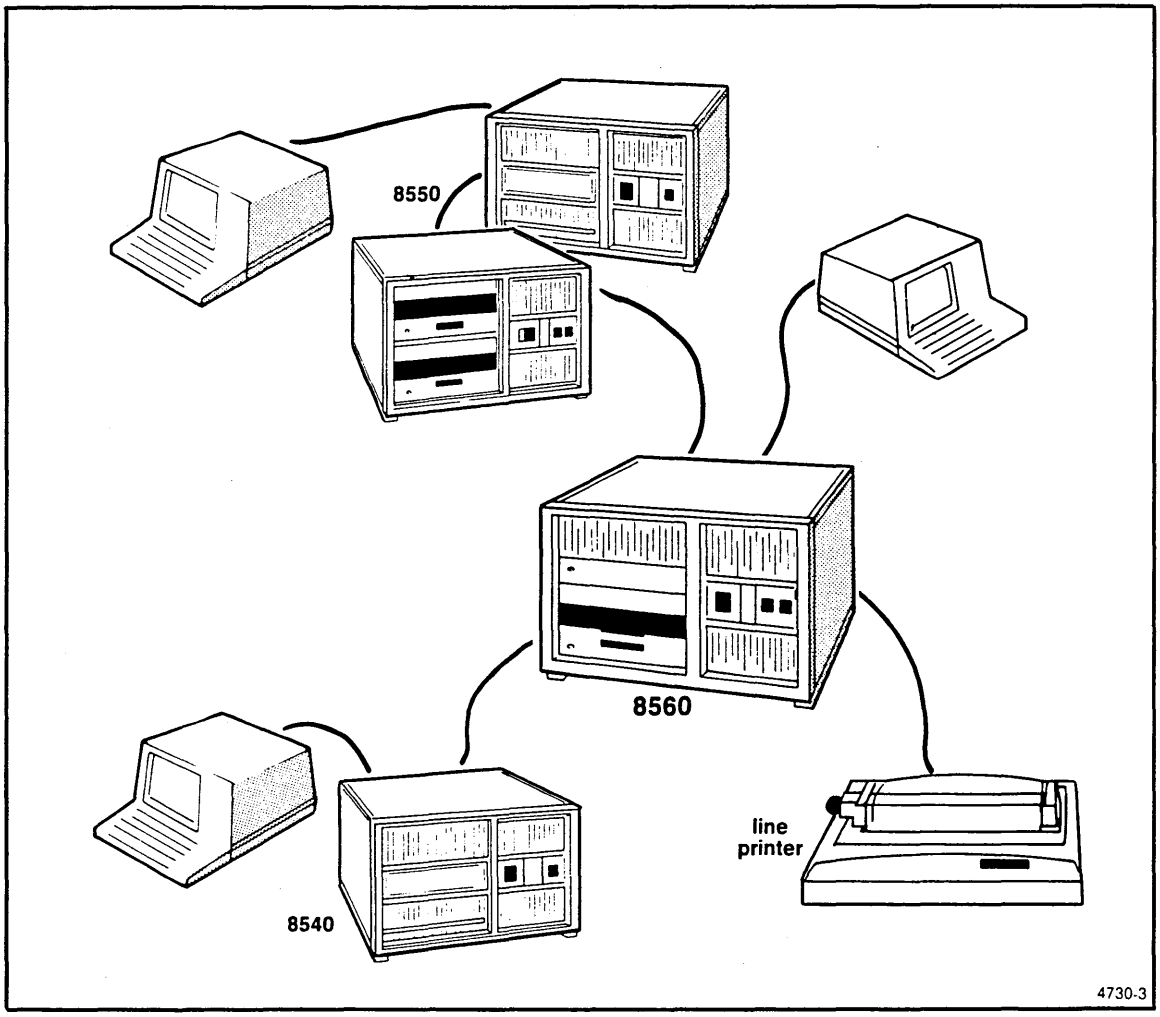


Fig. 1-2. A multi-workstation configuration for the 8560.

In this configuration, one terminal communicates directly with the 8560 and serves as a software development station. Two more terminals communicate with the 8560 through an 8550 and 8540, and serve as hardware/software integration stations. A line printer is also connected.

Minimum 8560 Series Configuration

A minimum 8560 Series configuration consists of an 8560 Series development system and a terminal. With the minimum system, you can create and maintain files and develop programs. After installing the appropriate software options, you can compile and assemble programs, create subroutine libraries, and link program modules into executable load modules.

The 8560 Series development system contains the following components:

- For an 8560, a 35.6-megabyte, fixed Winchester-technology disk drive. For an 8561, a 13.6-megabyte, fixed Winchester-technology disk drive. The disk contains the TNIX operating system, as well as optional software, workspace, and user files.
- A 1-megabyte flexible disk drive. The drive accepts disks that are either single-sided or double-sided, and either single-density or double-density. The disks supplied with your 8560 are double-sided and double-density. The flexible disk drive serves primarily to transfer programs and to back up files from the fixed disk.

The terminal is a CRT terminal or other RS-232-C compatible I/O device through which you communicate with the 8560. The terminal must have the full ASCII character set (including lowercase characters) and must run at 300, 600, 1200, 2400, 4800, or 9600 baud. The 8560 performs best with a TEKTRONIX 4105M or CT8500 terminal.

The terminal can access not only the 8560, but also any workstations that are attached to the 8560. In addition, the system terminal may attach directly to a workstation. For more information about system configurations, see the *Communication with 8540s and 8550s* section of this manual.

8560 Options

Figure 1-3 shows the options available for use with the 8560 system. The following text describes each option.

System Options

Line Printers. The 8560 accepts two RS-232-C line printers. Workstations attached to the 8560 can also be connected directly to line printers. The TEKTRONIX 4643 Line Printer is designed for use with the 8560.

Additional Memory. The 8560 comes with a standard 256K bytes of main memory. Optional 256K and 512K memory boards allow you to expand user-available memory. Any combination of one or two 256K-byte and 512K-byte boards may be used, providing a maximum 1M byte of memory.

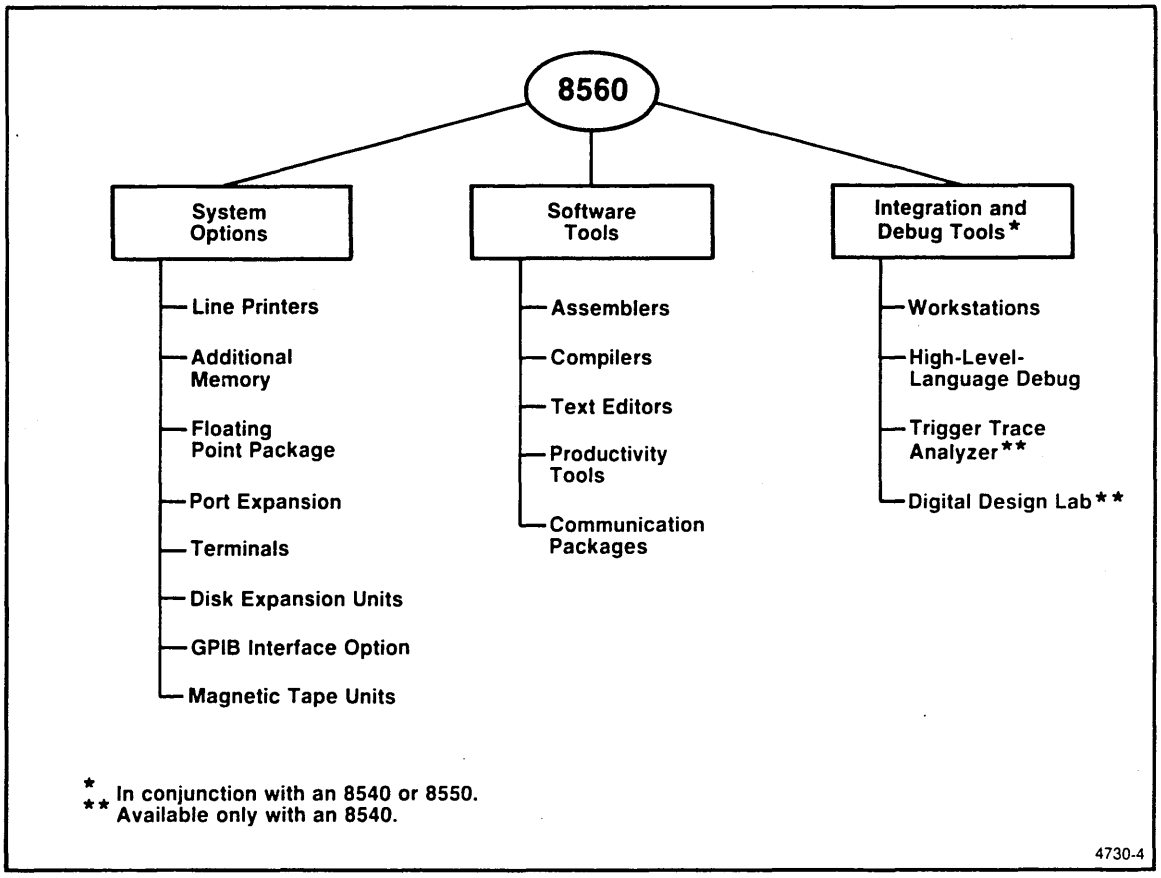


Fig. 1-3. 8560 system components.

The optional memory boards are recommended if your 8560 will be used by more than three users at the same time. Even if your system will not be used by more than three users at once, the additional memory can be used to improve system performance.

Floating Point Package. This option is an integrated circuit that increases the execution speed of many software packages. It is required for the optional Native Programming, Text Processing, and Auxiliary Utilities software packages, discussed later.

Port Expansion. The 8560 comes equipped with an I/O Processor board that controls I/O ports 0-3 on the rear panel. The Eight-User Upgrade option includes a second I/O Processor board that enables the 8560 to support up to four additional users (ports 4-7).

For the 8561, the standard I/O Processor board controls I/O ports 0-1 on the rear panel. Upgrade kits enable the 8561 to support either four or eight users.

Disk Expansion Units. The 8560 comes with a standard 35.6-megabyte fixed Winchester-technology disk drive. You can add up to three additional 35.6-megabyte TEKTRONIX 8503 Disk Expansion Units.

The 8561 comes with a 13.6-megabyte disk drive. In order to support any 8503 Disk Expansion Units, the 8561 must have the Four-User or Eight-User Upgrade option.

GPIB Interface Option. The GPIB Interface option allows you to use a tape drive for backing up files. The GPIB Interface is designed primarily for use with Dylon Corporation's Series 3 and Series 9 Magnetic Tape Systems.

Software Tools

Assemblers. An assembler translates assembly language source modules into machine language object modules. Tektronix provides an assembler for every microprocessor supported by the 8540 Integration Unit. The TEKTRONIX Series B Assemblers are intended for use with the 8560's TNIX operating system.

Every assembler option includes a linker and library generator. A linker combines object modules (produced by an assembler or compiler) into a load module, which may be loaded into memory and executed. A library generator enables you to create and modify libraries of commonly used routines. When you include calls to library routines in your program, the linker inserts the necessary object modules from the library into your load module.

Compilers. A compiler translates a high-level language (such as C or Pascal) into machine language or assembly language. Tektronix provides 8560 Pascal compilers for the Z8001/Z8002, 68000/68008, and 8086/8088 microprocessors.

Text Editors. In addition to the standard TNIX line-oriented editor *(ed)*, the TEKTRONIX ACE Screen Editor and Language-Directed Editor (LDE) are available for use on the 8560. Both ACE and LDE are screen-oriented editors. LDE can check a program for correct syntax, allowing you to correct errors while you're still in the editor.

Productivity Tools. Three optional command sets are available for the 8560:

The **Text Processing Package** contains the **nroff** and **troff** text processors, a table set-up program, an indexing command, a spelling error detection command, and several other capabilities.

The **Native Programming Package** offers compilers, an assembler, a debugger, archiving and library maintenance, lexical analysis, program generation, and compiler generation. These tools support programs that run on the 8560, rather than on an emulator in a workstation.

The **Auxiliary Utilities Package** consists of numerous commands, ranging from a desk calculator to a pattern-scanning and processing language to computer-aided instruction about TNIX.

Communication Packages. Communication packages allow the 8560 to exchange data with other systems. The UNICOM package enables an 8560 to communicate with other TNIX- or UNIX-based systems. The COMM package enables an 8560 to serve as host to Intel microcomputer development systems.

Integration and Debug Tools

Workstations. In combination with a workstation, the 8560 forms a complete development system, as described earlier in this section. The 8560 can serve as host to the 8540 Integration Unit, the 8550 Microcomputer Development Lab, and the 8001 and 8002 Microprocessor Labs.

High-Level-Language Debug. In addition to the symbolic debug commands included in workstation operating systems, TEKTRONIX Pascal Debug (PDB) lets you debug your software in terms of Pascal statements and data structures instead of assembly-level constructs.

Trigger Trace Analyzer. The Trigger Trace Analyzer is an 8540 option that allows you to monitor the buses and selected control signals in the prototype hardware while your program executes at normal speed.

Digital Design Lab. The Digital Design Lab (DDL) is a debugging tool that provides time correlation between data generated by the TEKTRONIX Trigger Trace Analyzer (TTA) and the TEKTRONIX Digital Analysis System (DAS). This allows you to analyze the software events (from the TTA data) that correspond in time to specific hardware events (from the DAS data).

Summary

Figure 1-4 shows an 8560 configuration that includes several options. Table 1-1 lists a number of TEKTRONIX products available for use throughout the design cycle, and the manual that describes each product.

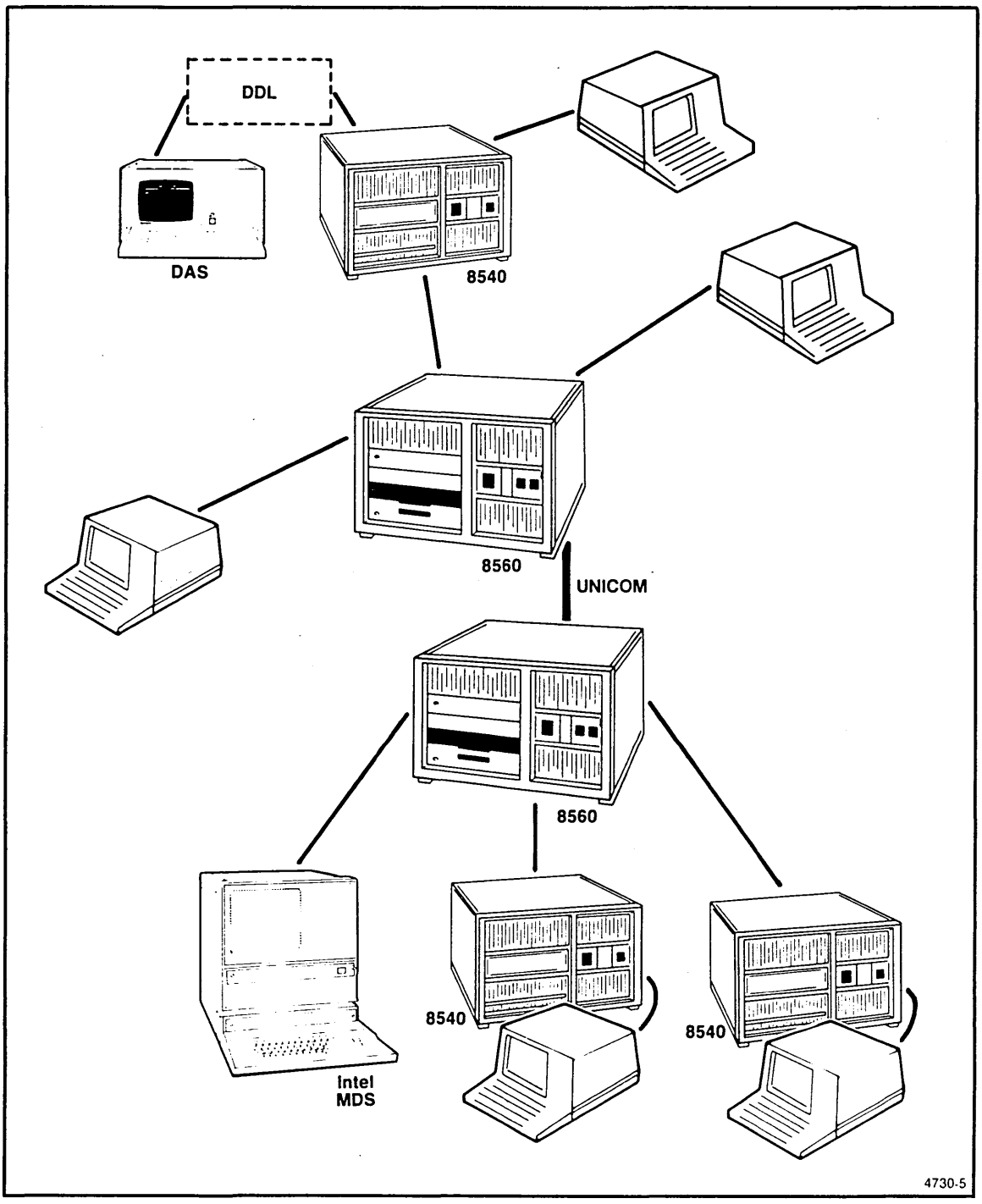


Fig. 1-4. An 8560 network.

In this configuration, two 8560s are linked via UNICOM. One of the 8560s supports an Intel MDS, along with two 8540s. The other 8560 communicates with a TEKTRONIX Digital Analysis System (DAS) by means of the Digital Design Lab (DDL).

Table 1-1
Using the 8560: A Guide to Products and Documentation

Development Task	Product	User Manuals
Create and edit design documents	Text Processing Package	8560 MUSDU Text Processing Package Users Manual
Create and edit source code	ACE Screen Editor	8560 MUSDU ACE Screen Editor Users Booklet
	Language-Directed Editor	8560 MUSDU Language-Directed Editor Users Manual
	Ed	8560 Series MUSDU System Users Manual
Assemble/Compile/Link	Series B Assembler	8500 Modular MDL Series Assembler Core Users Manual for B Series Assemblers (plus host- and microprocessor- specific supplements)
	Pascal Compiler and Integration Control System	8560 MUSDU Pascal xxxx Compiler Users Manual/8500 Modular MDL Series Pascal Language Reference Manual
	Linker	8500 Modular MDL Series Assembler Core Users Manual for B Series Assemblers
	Make utility program	8560 Series MUSDU System Users Manual
Debug	Pascal Debug	8500 Modular MDL Series Pascal Debug Users Manual
	8540/8550 debug commands	8540 or 8550 System Users Manual
	Digital Design Lab	8560 MUSDU Digital Design Lab System Users Manual
	Trigger Trace Analyzer	8500 Modular MDL Series Trigger Trace Analyzer Users Manual

GETTING STARTED

This subsection is intended to help you get started performing useful tasks on the 8560. This discussion covers the following topics:

- **Logging In.** Shows how to sign in to the 8560's TNIX operating system.
- **Using Keyshell.** Describes the Keyshell interface, which allows you to accomplish tasks by pressing function keys as well as by typing commands.
- **Entering Commands Directly.** Describes how to enter commands if your terminal does not support the Keyshell interface, or if you choose not to use Keyshell.
- **Tutorial.** Introduces you to a basic tool kit of TNIX and commands.

NOTE

The following discussion assumes that your 8560 has been installed and booted, and that the switches and options on your terminal are set appropriately. If this has not been done, refer to your ***8560 Series System Manager's Guide*** *and to your terminal users manual for instructions.*

Notation

This section uses the following notation conventions to illustrate how to enter commands:

- The $ prompt is shown before each command.
- Characters that you type are underlined. TNIX prompts and responses are not underlined.

Logging In

Once your 8560 is booted and your terminal is set appropriately, TNIX displays the login prompt on your terminal screen:

```
login:
```

On some terminals, you may need to press the RETURN key twice or press the BREAK key a few times in order to get the login prompt.

NOTE

If your terminal is connected to an 8540 Integration Unit or an 8550 Microcomputer Development Lab, you must turn on the terminal and the workstation before you log in to TNIX. Once you have booted the 8540 or 8550, enter TERM mode by issuing one of the following commands:

```
$ config term          [for an 8540]

$ config term t=7      [for an 8550]
```

These commands enable your terminal to speak directly with the 8560, while still communicating with your workstation.

Before you can communicate with TNIX, you must obtain a TNIX login name from your system manager. Your manager may also have assigned a password to you. In response to the login prompt, type your login name in lowercase, then press the RETURN key. For example:

```
login: smith
```

If a password is required, TNIX will ask you for it. For security reasons, TNIX will avoid displaying the password as you type it.

Once you've logged in, TNIX asks you to indicate what type of terminal you are using. TNIX should then display a prompt—a dollar sign "$"—signaling that the system is ready to accept commands. You may also receive a message of the day just before the prompt sign, or you may be notified that you have mail. (For information on how to read your mail, see the *Operating Procedures* section of this manual.)

Your terminal may also display a row of labels near the top or bottom of the screen. If you see such labels, you can use the Keyshell interface, which lets you accomplish tasks by pressing function keys as well as by typing commands. If you do not see these labels, skip the following discussion of Keyshell and go to the heading "Entering Commands Directly" later in this section.

Using Keyshell

Your terminal should now display a row of labels. Each label on the screen represents a task you can initiate by pressing the corresponding function key. On most terminals, the function keys are near the top of the keyboard. Figure 1-5 shows the Keyshell display and the function keys on a TEKTRONIX 4105M terminal.

Try pressing one of the function keys. A new set of key labels will appear, each representing a related new task or set of tasks.

Some of the function keys simply move you from one related set of labels to the next, but most keys build TNIX commands for you and submit them to TNIX for execution as if you had typed them yourself. The commands are displayed on the screen as they are built and executed.

Experiment with Keyshell to find out what it can help you do. Keep the following in mind as you work:

- To return to the previous set of key labels, press the rightmost key (labeled *done*).
- To cancel a command that Keyshell is building for you, press *done*—the command will be erased before it has executed.
- To stop a command that has already started to execute, type a CTRL-C—hold down the key marked "CTRL" (for "control") and type a "c".
- To get information about a particular TNIX command, press **shifted** function key 1 for online help.

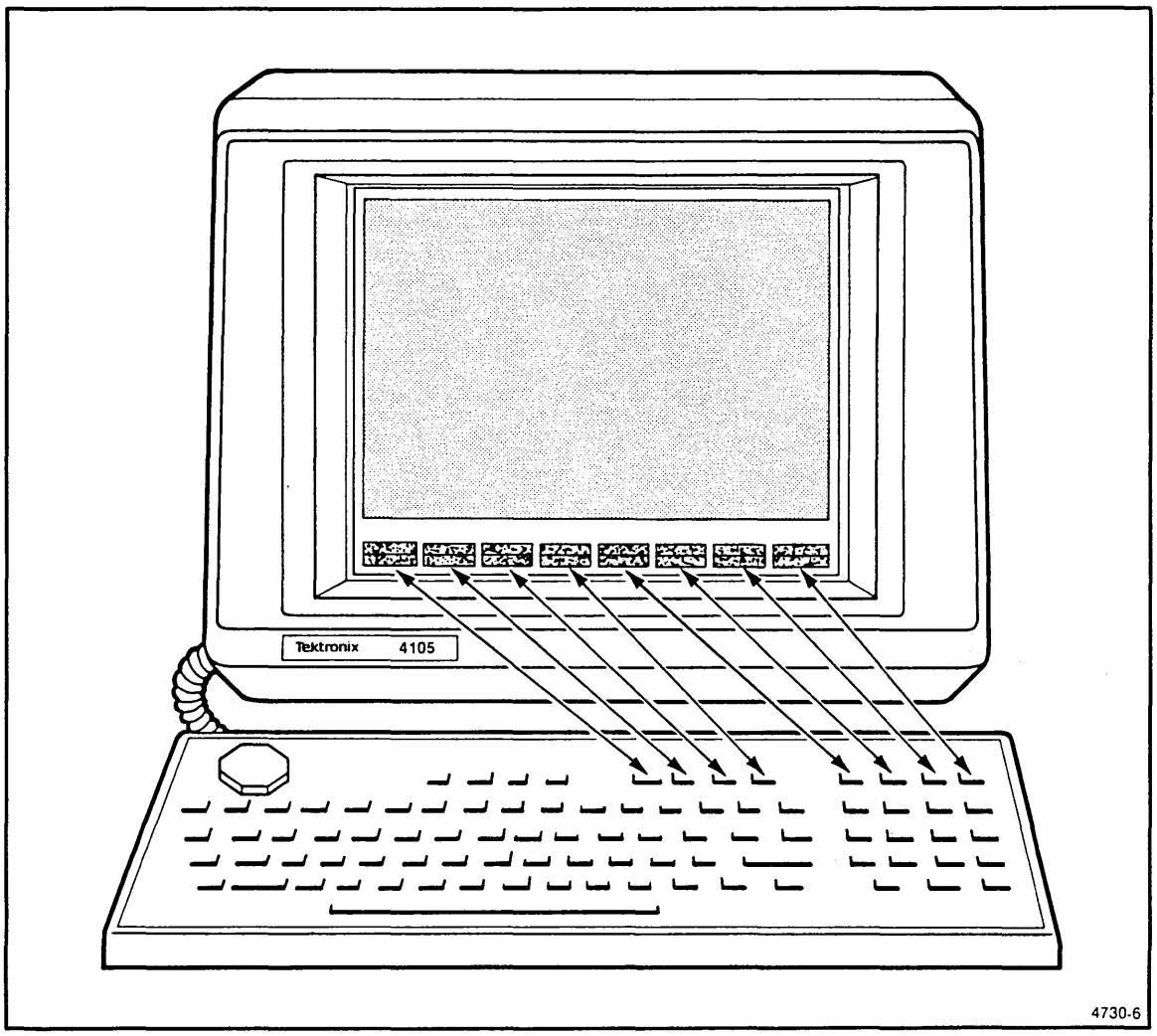


Fig. 1-5. Keyshell labels and function keys on TEKTRONIX 4105M terminal.

Shifted Function Keys

With the unshifted or "lowercase" function keys, the key labels change as you travel through a "tree" of key labels. However, the shifted or "uppercase" function keys perform functions that are useful no matter where you are in the tree, and their meanings do not change.

On the 4105M and CT8500 terminals, these keys are labeled by a plastic keyboard overlay instead of an on-screen display. On other terminals, the keys may not be labeled. Table 1-2 summarizes the shifted function keys.

Table 1-2
Keyshell Shifted Function Keys

Key	Label	Function
1	Help	Gives you access to online information about TNIX commands.
2	Where Am I	Shows you which function keys you pressed most recently.
3	Expand Keys	Shows the full text of any key labels that are too long for the label area.
4	Explain Key Labels	Describes the tasks that the current set of labeled keys can perform.
5	Redraw Screen	Redisplays the function key labels if you have inadvertently erased them.
6	History Fwd	A companion to History Back, this key scrolls through your command history in the opposite chronological direction.
7	History Back	With repeated presses, provides a list, in chronological order, of the TNIX commands you have typed or issued via function keys. When you reach a command you want to execute again, press the RETURN key.
8	Execute Last Command	Ordinarily, repeats your last command. If you have started to type a new command, this key searches for a previous command that began similarly and completes the command line with that text.

The command history keys also activate a command history editor. When the history editor is active, the unshifted function keys are labeled with editing functions that let you change the command line currently on display. The *Keyshell* section of this manual describes command history and the command history editor in more detail.

Mixing Keyshell Functions with Typed Commands

You are always free to enter commands by typing them literally instead of by pressing function keys. In fact, as you learn more about TNIX you may find that a mixture of the two kinds of command entry is most efficient for you. For more information about typed commands, read the following discussion, "Entering Commands Directly".

A few TNIX commands may not work as expected while you're using Keyshell. These exceptions are described in the Keyshell section of this manual.

Ending Your Keyshell Session

To end your Keyshell session, press *done* until you arrive at the top level of keys (key 8 will be labeled *exit*). Press *exit* and TNIX will ask whether you want to log out or to use the regular TNIX command language without Keyshell assistance. Press *log out* if you want to end your work session.

If you inadvertently leave Keyshell, type **ksh** to return.

Entering Commands Directly

The following paragraphs tell how to enter TNIX commands. Read this material if your system does not support Keyshell, or if you want to type commands while you're using Keyshell. This material does not describe the TNIX command language. For that information, refer to the tutorial that follows this subsection, and to the *TNIX Operating System* and *Operating Procedures* sections of this manual.

General Information

You may enter a command whenever you see the "$" prompt. The command is not read and processed by TNIX until you press the RETURN key.

Selecting the 8540 or 8550

If your terminal is attached to the 8560 and you also want to communicate with a workstation, you must specify the HSI I/O port to which the workstation is attached. To do this, enter the following commands:

```
$ IU=n; export IU          [for an 8540]

$ stty IU >/dev/ttyn       [for an 8550]
$ IU=n; export IU
```

In place of **n**, enter the number of the HSI I/O port (from the 8560's rear panel) to which the 8540 or 8550 is attached.

Do not attempt to communicate with the 8540 or 8550 while they are in LOCAL mode.

Online Help Tools

Two TNIX commands allow you to obtain information about TNIX while you are logged in:

- The **man** command, followed by a command name, displays an online "manual page" that describes the command. For instance, **man ls** displays information about the **ls** command.
- The **index** command works like the index of a book. If you type **index** followed by one or more keywords, TNIX lists the online manual pages that contain information about that keyword or combination of keywords. For example, **index directory** lists the online manual pages that discuss directories.

Mistakes in Typing

If you notice a typing mistake on a line before you press the RETURN key, there are two ways to recover:

- The BACKSPACE key erases the last character typed. Successive backspaces will erase all the way to the beginning of the line, but not beyond. (CTRL-H is identical to a backspace. Type "h" while holding the CTRL key down.)
- CTRL-U erases all of the characters typed on the current line. If the line is irretrievably garbled, type CTRL-U and start the line over.

If you don't notice a mistake until after you've pressed the RETURN key, you can either retype the command or use CTRL-K, which retypes successive characters of the previous line. For example, assume you typed **pwx** instead of **pwd** (the print working directory command). If you type CTRL-K twice, the "p" and "w" are retyped for you. You can then type the "d" and press the RETURN key, and the command will be executed correctly. (Note: CTRL-K does not work when you are using Keyshell.)

Stopping a Program

You can stop most commands by typing a CTRL-C.

When your workstation is in TERM mode, do not toggle the RESTART switch on the workstation's front panel. If you need to interrupt system operation, type CTRL-C.

If you restart the workstation while it is in TERM mode and workstation commands are still active on the 8560, then the workstation and the 8560 may not be able to resume communication. If your system seems to hang after you restart the workstation, you or your system manager must kill the workstation commands from a different terminal.

Logging Out

If your terminal is connected directly to the 8560, you can log out by typing the **logout** command. You can also type **login**, which logs you out and prepares the terminal for someone else to log in. Turning off the terminal may or may not log you out.

If your terminal is connected to an 8540 or 8550 workstation in TERM mode, log out by typing **config local; logout** on the same line. These commands switch the 8540 or 8550 to LOCAL mode and log you out of TNIX.

Tutorial

The following tutorial shows you how to use a number of common TNIX features and commands. You can perform these tasks by typing the commands shown or by using the Keyshell function keys.

Create a File

When you need to create a text file—such as a program, a memo, or a specification—you will normally use a text editor to enter the text, and will store the text in an 8560 file. (A file is simply a collection of information stored in the 8560.)

For now, however, we'll use a shortcut to create a file—the TNIX **cat** command. **Cat** is ordinarily used to display the contents of a file or to merge two or more files, but you can also use it to store text in a file. Try entering the following text:

```
$ cat >nonsense
As I was standing in the street
As quiet as can be,
A great big ugly man came up
And tied his horse to me.
<CTRL-D>
```

After you type the CTRL-D, a file called *nonsense* is created; it contains four lines of text.

List a File

Now use the **ls** (list) command to verify that the file exists:

```
$ ls
nonsense
```

TNIX displays a list of the files in the current directory, sorted into alphabetical order. Other variations are possible, however. For example, the command **ls -t** lists files in the order in which they were last changed, most recent first.

The use of optional arguments ("flags") that begin with a hyphen (like **–t**) is a common convention for TNIX commands. With any command, you can combine flags or options. In general, if a command accepts flags, they precede any filename arguments. It is important to separate the various arguments with spaces: **ls–t** is not the same as **ls –t.**

The **ls** command—like many other TNIX commands—has a number of options. For a full description of what **ls** enables you to do, enter **man ls**.

Copy a File

Now try making a *copy* of the file *nonsense*. Enter the following command to create a copy that contains the same text but is named *funnyfile*:

```
$ cp nonsense funnyfile
$
```

The only system response is the return of the "$" prompt after the file has been copied. If you want to verify that the file was copied, enter **ls** again:

```
$ ls
funnyfile
nonsense
```

View the Contents of a File

Once you've created a file, you can use one of several commands to view the text that it contains.

The **cat** command displays on your terminal the contents of all the files named in a list. For example, to view *nonsense*, enter

```
$ cat nonsense
```

To view both of your files, enter

```
$ cat nonsense funnyfile
```

The two files are simply merged end-to-end or con**cat**enated onto the terminal.

The **pr** (print) command prepares a file to be printed on a line printer, but displays the file on your terminal. A file processed by **pr** includes headings with date, time, page number, and file name at the top of each "page", plus extra lines to skip over the folds in lineprinter paper.

You can send the formatted file to the line printer by **piping** the output of the **pr** command to the **lpr** command. (The output of any TNIX command can be automatically sent to another command.)

```
$ pr nonsense ¦ lplr
```

The pipe symbol "¦" sends the formatted file *nonsense* to line printer 1.

Rename a File

The TNIX **mv** command "moves" a file by giving it a new name. Enter the following command to rename *nonsense*:

```
$ mv nonsense serious
```

The file is now named *serious*. Enter the **ls** command again to verify:

```
$ ls
funnyfile
serious
```

CAUTION

If you move a file to another one that already exists, the previous contents are lost forever.

Explore the File System

When you first created the file *nonsense*, how did the system know that there wasn't another *nonsense* somewhere else, especially since the person at the next workstation may also be reading this manual? The answer is that each user has a private **directory** that contains only the files that belong to him or her. When you log in, you are "in" that directory, which—logically enough—is called your "login directory" or HOME directory. ("HOME" is in uppercase letters because it's an "environment variable". You'll learn more about environment variables in the *TNIX Operating System* section of this manual.)

Unless you take special action when you create a new file, the file is placed in the directory that you are currently in; this is normally your own directory, so that the file is unrelated to any other file of the same name that might exist in someone else's directory.

TNIX organizes all files into a **tree**, with your files located several levels down from the *root* directory at the top of the tree. Figure 1-6 shows a portion of the TNIX file tree. You can move around this tree and find any file in the system by traveling along the proper set of branches.

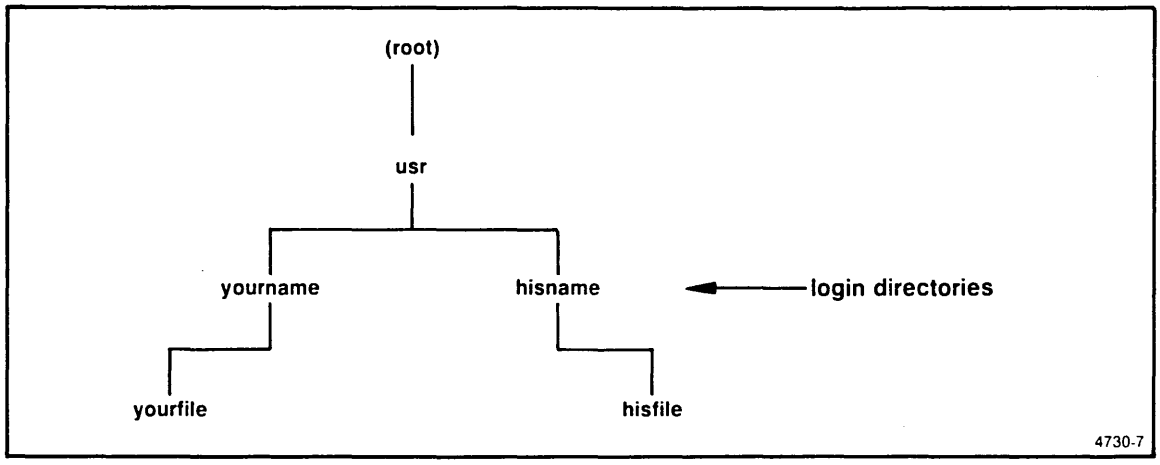


Fig. 1-6. A portion of the TNIX file tree.

Other users can view your files by entering the command

```
$ cat /usr/yourname/yourfile
```

In this example, the file is specified by its "absolute pathname"—*/usr/yourname/yourfile*. The absolute pathname traces the file's location, beginning with the root directory and ending with the filename itself. A slash ("/") is used to represent the root directory and to separate components of the pathname.

You can find out what files your neighbor has by entering

```
$ ls /usr/hisname
```

You can make your own copy of one of his files:

```
$ cp /usr/hisname/hisfile yourfile
```

Of course, your friend may not want to share his files. He may have altered the **protection modes** to deny other users access to his files. See the discussion "File Protection" in the *TNIX Operating Procedures* section of this manual for a description of file protection modes and how to set them.

Change Your Current Directory

The **cd** command moves you to another directory, which may be another user's directory or a subdirectory that you've created under your login directory. (Again, you can move to another user's directory only if she or he has not denied you permission to enter the directory.) For example, the following command moves you to another user's login directory:

```
$ cd /usr/hername
```

Now, when you use a filename with a command like **cat** or **pr**, the pathname refers to the file in your friend's directory.

If you forget what directory you are in, type **pwd** (print working directory) to find out.

Create a Directory

As you continue to generate files, you will probably find it convenient to create a subdirectory to store related files, rather than keeping them all in one large directory. (It's similar to keeping all your tax records in an organized filing system, rather than in a shoe box.)

For example, you might create a directory called *trivia* to store *serious* and *funnyfile*:

```
$ mkdir trivia                          [create the directory]
$ cd trivia                             [move to the directory]
$ mv /usr/yourname/serious serious      [move serious to the current directory]
$ mv /usr/yourname/funnyfile funnyfile  [move funnyfile to the current directory]
```

Use the Pattern-Matching Characters

Enough of funny filenames. Let's assume now that you are writing a complex system specification. Your document consists of a number of sections and subsections, so you type the document as a number of files: *sect1*, *sect2*, *sect3*, etc.

There are advantages to a systematic naming convention. You can tell at a glance where a particular file fits into the whole. And if you need to print the entire document, you don't have to specify each file name:

```
$ pr sect1 sect2 sect3 sect4 sect5 sect6 sect7 | lplr
```

Instead, you can use **pattern-matching characters** to simplify your task. For example:

```
$ pr sect* | lplr
```

The asterisk "*" matches "anything at all", so this command prints all files whose names begin with *sect*. The "*" can appear anywhere in the filename and can occur several times. Thus, **rm *sect*** removes all files that contain the characters *sect* as any part of their name.

In addition, "*" by itself matches every filename. Thus, **pr *** displays all the files in the current directory, and **rm *** removes **all** the files in the current directory. (Use this one with caution!)

The asterisk is not the only pattern-matching feature available. Square brackets ([ ]) match any single character inside the brackets. Thus, the folowing command prints sections 1, 2, 3, 4, and 7:

```
$ pr sect[12347] | lplr
```

A range of consecutive letters or digits can also be abbreviated within brackets, so the following command also prints sections 1, 2, 3, 4, and 7:

```
$ pr sect[1-47] | lplr
```

(Note that the 4 and 7 are **not** the two-digit number, 47.) Letters can also occur within brackets: [a-z] matches any character in the range "a" through "z".

Finally, a question mark matches any single character. Thus, **ls ?** lists all files that have single-character names, and **ls sect?** lists *sect1*, *sect2*, and so forth.

You can "turn off" the special meaning of "*", "?", and "[...]" by enclosing the entire argument in single quotes. For example, the command **ls '?'** lists information about a file named **?**.

The *TNIX Operating System* section of this manual contains a table that summarizes information on the pattern-matching characters.

Send Output to Files Instead of the Terminal

Most of the commands you have seen so far produce output on the terminal. For all TNIX commands, however, a file may replace the terminal for both input and output. For example, **ls** displays a list of files on your terminal. But if you type **ls >filelist**, the list of your files is placed in the file *filelist*, which will be created if it does not already exist, or overwritten if it does. The symbol ">" means "place the output in the following file, rather than on the terminal." Only error messages will appear on the terminal. You can display this listing by entering **cat filelist**.

As another example, you can combine several files into one larger file by sending the output of the **cat** command to a file:

```
$ cat file1 file2 file3 >file123
```

The symbol ">>" operates very much like ">", except that it **appends** the information—adds it to the end of the specified file. That is,

```
$ cat file1 file2 file3 >>file123
```

concatenates the three files to the end of whatever is already in *file123*, instead of overwriting the existing contents of *file123*. As with ">", if *file123* does not exist, TNIX creates it for you.

Summary

This tutorial has introduced you to the TNIX file system and TNIX command entry. Table 1-3 lists the commands and symbols covered in this discussion.

Table 1-3
Summary of Tutorial

Commands/Symbols	How Used
cat	Create a file, display a file.
ls	List contents of a directory.
cp	Copy a file.
pr	Format a file for printing.
lp1r	Send a file to line printer 1.
mv	Rename a file, move a file to a new directory.
cd	Move to another directory.
mkdir	Create a new directory.
¦	Pipe symbol: connect the output of one command to the input of another command.
* [ ] - ?	Pattern-matching characters.
>	Send the output of a command to a file rather than to the terminal.
>>	Append the output of a command to a file.

FOR CONTINUED LEARNING

This Learning Guide has given you an overview of the 8560 system, and showed you how to get started using the 8560. The following paragraphs tell you where to get more information on particular aspects of the 8560 and its TNIX operating system.

The TNIX Operating System

Section 2 of this manual, *TNIX Operating System*, contains information designed to give you an understanding of the basics of the TNIX operating system.

Section 3, *Operating Procedures*, uses a "cookbook" approach to performing tasks on the 8560. Like Keyshell, this section makes it easy to get your work done without knowing a lot about the intricacies of the operating system.

Section 11, the *Glossary*, defines new terms.

TNIX Command Language

Section 9 of this manual contains an abbreviated dictionary of standard TNIX commands. In addition, each command is fully described in an online manual page. To view the manual page for a particular command, enter the **man** command, followed by the command name. For example, enter **man ls** to see the manual page for the **ls** command.

If you frequently refer to particular manual pages, you may find it worthwhile to print a copy of them. To print a manual page on line printer 1, enter the following command:

```
$ man command | pr | lplr
```

In place of **command**, enter the name of the command you want to print.

The optional *8560 Series MUSDU System Reference Manual* provides a detailed description of each TNIX command.

The TNIX Editor

Section 5 of this manual describes the TNIX editor, *ed*.

System Maintenance

For any multi-user system, it's generally most efficient to designate one person the "system manager" and make this person responsible for tasks such as installing software and creating user accounts. The *8560 Series System Manager's Guide* describes the tasks that an 8560 system manager needs to perform, and tells how to perform them. This manual also shows how to install an 8560 and how to add workstations and peripherals.

Intersystem Communication

If you have any workstations attached to your 8560, you will need to learn more about communication between a terminal, a workstation, and an 8560. Section 7 of this manual, *Communication with 8540s and 8550s*, presents general information about communication between an 8560 and workstations. The *8560 Series System Manager's Guide* covers certain hardware and software configuration procedures that must be performed by the system manager before communication can take place.

The *System Users Manual* for your 8540 or 8550 offers additional information about intersystem communication, dealing especially with the LOCAL mode configuration. In that mode, your terminal communicates directly with the workstation, with no mediation from the 8560 TNIX operating system.

The *Emulator Specifics Users Manual* for each emulator includes a demonstration run that shows how to create, assemble, link, and execute a program using the 8560 with either an 8540 or 8550 and an emulator.

Shell Programming

The shell is the program that interprets what you type as commands and arguments. The shell also constitutes a *programming language*: in addition to the TNIX commands, the shell programming language includes high-level control statements (**for** and **while** loops, **case** and **if** statements), parameters, variables, subroutines, string substitution, and error handling. You can create your own commands by placing command sequences in files called *shell procedures*.

Section 2 of this manual, *TNIX Operating System*, describes general shell capabilities. Section 4, *Shell Programming*, explains how to write shell procedures.

Maintaining Files

The **make** utility program, described in Section 6 of this manual, is a command generator: it issues commands to maintain and update a set of program files. Whenever any part of a program is changed, **make** regenerates the proper files simply and correctly. With moderately complex programs, this service can be an almost indispensable time-saver and program bug-saver.

You can also use **make** to update other files and to perform routine tasks such as printing.

Section 2
TNIX OPERATING SYSTEM

ILLUSTRATIONS

TABLES

Section 2

TNIX OPERATING SYSTEM

OVERVIEW

Section 1 of this manual presented a general overview of the 8560 Series system, and showed you how to log in to TNIX and enter TNIX commands. This section provides an overview of the TNIX operating system, and includes discussions that explain the TNIX file system and command language and show how to modify your TNIX environment and create your own commands. This information should be useful whether you use the Keyshell function keys to enter commands, or construct and type the commands yourself.

TNIX FILE SYSTEM

This subsection describes the TNIX file system, including the following topics:

- **Files and Directories**. Defines the terms "files" and "directories", and shows how you can use directories to logically group your files.
- **Pathnames**. Describes pathnames, the current directory, and ways to abbreviate pathnames.
- **Filenames**. Gives the rules for forming valid filenames, and shows how you can use "wildcard" characters to abbreviate filenames.
- **Links to a File**. Shows how you can use the **ln** command to place a file in more than one directory.
- **File Protection**. Describes file protection modes and how to use them to control access to your files.
- **System File Structure**. Illustrates the standard TNIX directory hierarchy.

Files and Directories

A file is a set of related information that can be referred to collectively by one name. TNIX files are unstructured, and may be either text files or binary files. A source file, for example, is a text file that contains the source code for a program or routine.

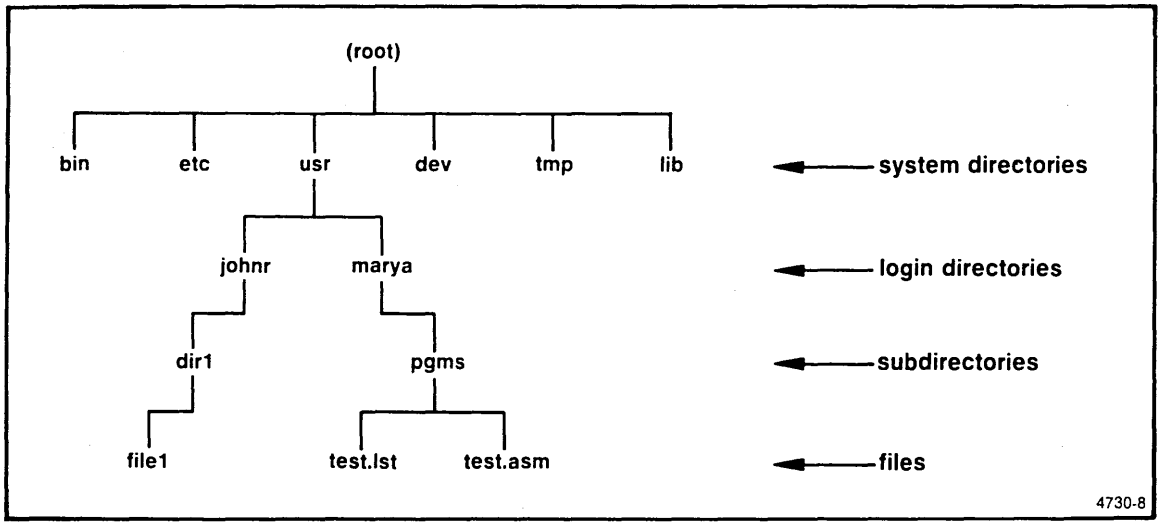


Fig. 2-1. The TNIX file system.

TNIX stores files in a tree structure, with related files grouped together in a directory. Each user has a directory under the **usr** directory, and may create files and subdirectories.

In addition to "regular files", TNIX uses special files for each I/O device. This means that commands can easily take their input from and send their output to devices such as line printers and storage devices.

A directory is a file that consists of a list of filenames, with a pointer to each file or subdirectory contained within that directory. Directories make it easy to group related files together.

TNIX creates and maintains all files in a inverted tree structure, with the **root directory** at the top of the tree. Figure 2-1 illustrates the TNIX file structure. For information on the system directories, see the discussion "System File Structure" later in this section.

When you log in, TNIX places you in your "HOME directory" or "login directory", */usr/yourname*. Until you create subdirectories, all your files are placed in your HOME directory. Just as TNIX organizes the system files into several directories, you will probably find it practical to create a separate directory for each of your major projects and areas of interest. Figure 2-2 shows how one user has arranged his files. User joeb has created (with the **mkdir** command) three main directories: *proj1*, *proj2*, and *letters*. Under *proj1*, he has created a file (*docs*) and a subdirectory (*paspgm*). *Proj2* likewise contains a file and a subdirectory (*asmpgm*). The *letters* directory contains only files.

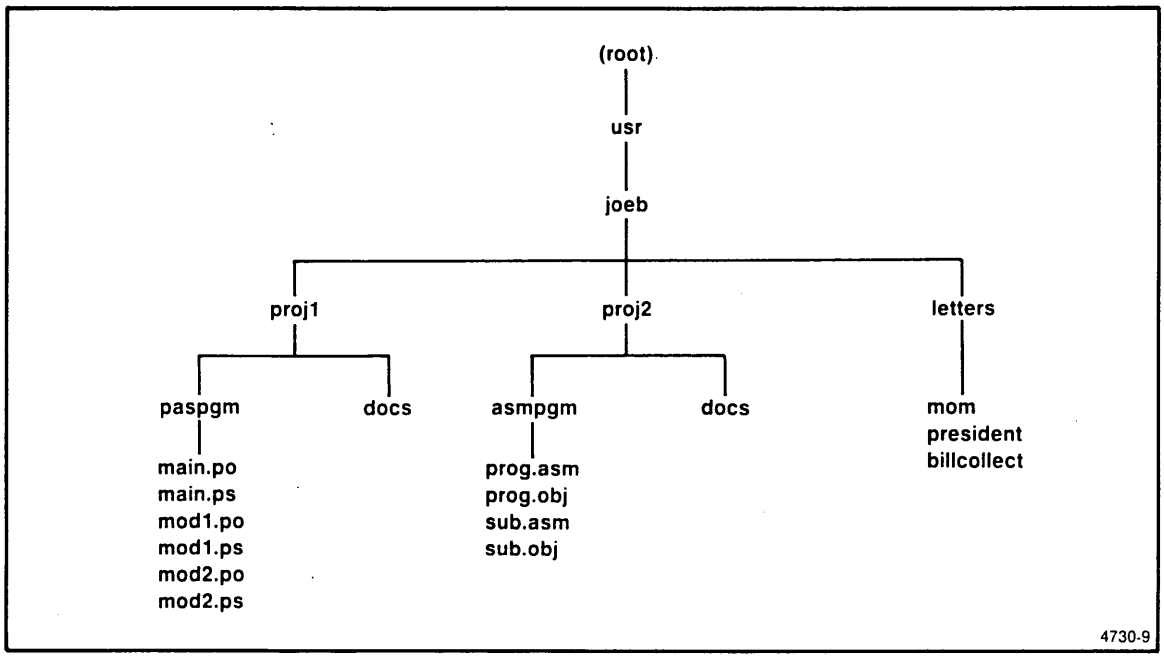


Fig. 2-2. A sample directory.

User joeb has three main directories, each containing one or more files or subdirectories.

Pathnames

A pathname specifies a unique name for each file or directory on the system. Pathnames may be either absolute or relative.

An absolute pathname (or full pathname) specifies a path from the root directory, and shows each of the file's "ancestors". A slash (/) is used to represent the root directory and to separate each part of the pathname. In Fig. 2-2, the absolute pathname for the file *main.ps* is */usr/joeb/proj1/paspgm/main.ps*. Note that joeb has two files called *docs*; however, their absolute pathnames are unique.

A relative pathname specifies a path from the current directory to another directory or file. Since a relative pathname starts from the current directory rather than from root, the pathname begins with the name of the next descending file or directory, rather than with the "/" character. For example, when joeb is in his *proj1* directory, the relative pathname of *main.ps* is *paspgm/main.ps*.

A pathname may also be a simple filename. For example, when joeb is in the the *letters* directory, the relative pathname of the file *president* is simply the filename: *president*.

The Current Directory. As you move around the file tree, the directory you are "in" at any given time is the **current directory** or **working directory**. The **pwd** (print working directory) command displays the absolute pathname of the current directory.

Pathname Abbreviations. TNIX provides two abbreviations for frequently-used pathnames:

- A period (.) refers to the current directory.
- Two periods (..) refer to the current directory's parent—the directory's immediate ancestor in the file tree.

Moving to a New Directory. The **cd** (change directory) command moves you around in the file tree. Assume that joeb (Fig. 2-2) logs in and decides to use **cd** to practice moving around his file tree.

```
$ pwd
/usr/joeb                 [he starts in his HOME directory]
$ cd letters              [moves to /usr/joeb/letters]
$ cd ../proj2             [moves to /usr/joeb/proj2]
$ cd /usr/joeb/proj1      [moves to /usr/joeb/proj1]
$ cd paspgm               [moves to /usr/joeb/proj1/paspgm]
$ cd..                    [moves to its parent directory, /usr/joeb/proj1]
$ cd                      [and returns to his HOME directory]
```

(These examples illustrate various ways of constructing pathnames; they don't necessarily show the most efficient ways to move around the file tree.)

Filenames

This subsection tells how to choose valid filenames (including how to use "special characters" in filenames), and shows how you can use "wildcard" characters to specify a file or group of files.

Valid Filenames

The following rules apply to filenames:

- A filename can be any length; however, TNIX stores only the first 14 characters.
- Uppercase and lowercase letters are distinct.
- A filename should not contain spaces, tabs, slashes, or control characters (such as CTRL-C).
- A filename that begins with a period is not displayed by most forms of the **ls** command, but is displayed by **ls -a**.

The following are valid TNIX filenames:

PasM (different from "pasm")
.utility
8.p
averylongfilename (truncated to "averylongfilen")

Special Characters

The following characters have a particular meaning to TNIX, and may cause problems if used in a filename:

```
/    \    &    ;    :    ?
!    -    $    *    "    >
<    [    ]    (    )    `
'
```

You can disable the special significance of these characters by one of the following methods:

- The backslash (\) disables any special character.
- Double quotes ("...") disable all special characters except the following:

  $ ` \ "
- Single quotes ('...') disable all special characters except a single quote.

Pattern Matching

The *Learning Guide* section of this manual briefly discussed TNIX's "wildcard" characters, which you can use to match one or more characters in a filename. Table 2-1 summarizes the TNIX wildcard characters; the following paragraphs show some examples.

Table 2-1
Pattern-Matching Characters

Character	How Used[a]
?	Matches any single character.
*	Matches any string of 0 or more characters, excluding "/".
[x..y]	Matches any one character in the set specified by x..y.
[x-y]	Matches any character in the range x-y.

[a] **A period (.) at the beginning of a filename must be explicitly matched.**

Examples. Assume that the current directory contains the following files:

```
main.asm
main.pl
main.po
main.ps
mod1.po
mod1.ps
mod2.ps
mod3.ps
```

You can use the **echo** command to see what patterns are matched by the wildcard characters. (**Echo** merely displays its arguments after they have been preprocessed by the shell.) Here are some examples:

echo main.p?	Echoes main.pl, main.po, main.ps.
echo *.ps	Echoes main.ps, mod1.ps, mod2.ps, mod3.ps.
echo mod[1-2]*	Echoes mod1.po, mod1.ps, mod2.ps.
echo main.p[m..z]	Echoes main.po and main.ps.
echo *	Echoes all files in the current directory.

Links to a File

The **ln** command enables a file to appear in more than one directory, possibly under more than one filename. Unlike **cp**, which creates a physical copy of the file, **ln** merely creates a new **path** to the file, as shown in Fig. 2-3. Linking thus saves disk space.

You can use linking in several ways:

- To share files with other users. Any change to the file is reflected in each "version" of the file, because all "versions" are in fact the same file. Linking thus ensures that each user has an up-to-date version of the shared file.
- To back up important files. All links to a file have equal status, and a file is removed only when all links to the file have been removed. If you accidently delete a file, any other links to the file still remain, and the file itself is not deleted.

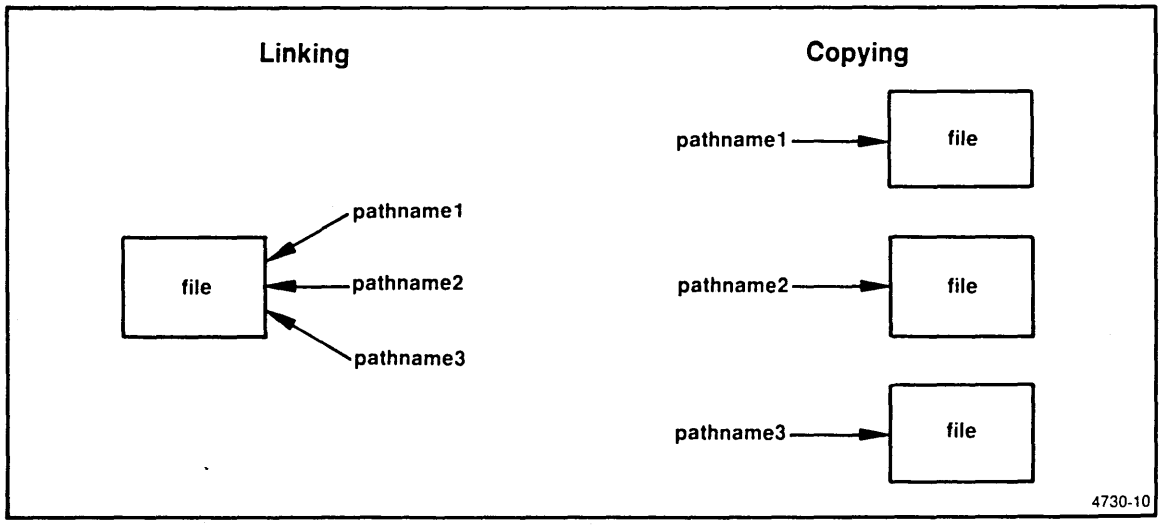


Fig. 2-3. Linking and copying a file.

Linking saves file space, because only one copy of the file exists.

Example

Let's assume that user joeb (from Fig. 2-2) decides to create a backup directory (*projects.bak*) and add links from that directory to *main.ps* and *prog.asm*. Starting from his home directory, joeb first creates the new directory and moves to it:

```
$ mkdir projects.bak
$ cd projects.bak
```

He then links to the two files, and lists the directory contents to verify the results. (Note that he uses the ".." abbreviation in the pathname when he enters the **ln** command.)

```
$ ln ../proj1/paspgm/main.ps
$ ln ../proj2/asmpgm/prog.asm
$ ls
main.ps
prog.asm
```

The absolute pathnames for the resulting links are */usr/joeb/projects.bak/main.ps* and */usr/joeb/projects.bak/prog.asm*. Figure 2-4 shows the new directory structure.

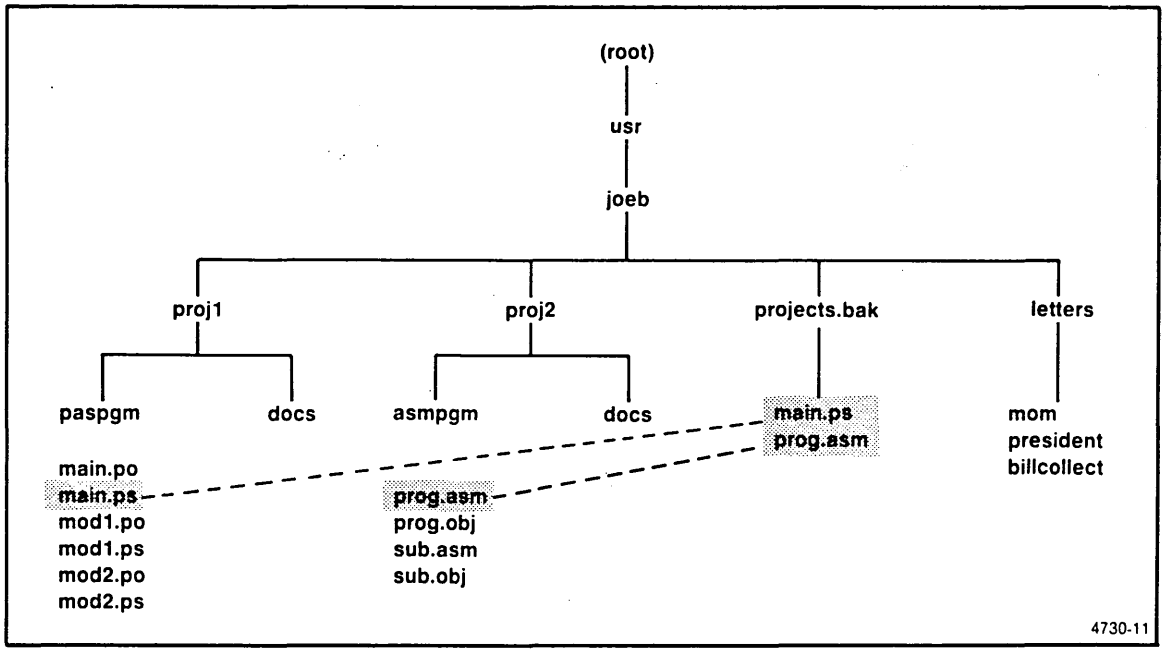


Fig. 2-4. Links to a file: modified sample directory.

The new directory *projects.bak* contains links to files in the *paspgm* and *asmpgm* directories.

Number of Links

The listing produced by the **ll** command shows the number of links to a file. In the following example, each file has two links to it:

```
$ pwd
/usr/joeb/projects.bak
$ ll
-rw-r--r-- 2 joeb    682 Feb 11 08:17 main.ps
-rw-r--r-- 2 joeb    220 Feb 11 08:25 prog.asm
```

Restrictions

You may not link to a directory. Also, you may not link across filesystems. (Filesystems are important only when you have one or more Disk Expansion Units installed. See the *Glossary* of this manual for a definition of filesystems.)

File Protection

TNIX has file protection modes that allow you to specify who can access your files and directories. By setting the protection modes, you can share files with other users, or protect files from access by other users. The following paragraphs describe the file protection modes.

Categories of Users

You can control access for three categories of users, as shown in Table 2-2.

Table 2-2
Categories of Users for File Protection Modes

Category	Abbreviation	Description
User	u	File's owner
Group	g	Members of the owner's group
Other	o	Other users: anyone on the system

When the system manager creates an account for a new user, he or she may assign the user to a **group**. For example, if several users are designing a microprocessor-controlled razor, the system manager might create a group called *razor* and assign each team member to this group. By setting the group protection modes, each group member can allow other members to access his or her files.

The */etc/group* file lists the groups that exist on your system (along with the members of each group). For information on how to create a group, refer to the *8560 Series System Manager's Guide*.

Types of Permission

You can specify three types of permission—read, write, and execute. Table 2-3 shows what the given permission allows you to do.

Table 2-3
Read, Write, and Execute Permission

Permission	File	Directory
Read	Permits you to read the file.	Permits you to list the contents of the directory.
Write	Permits you to create or modify the file.	Permits you to create or remove a file in the directory, link to a file in the directory, remove the directory.
Execute	Permits you to invoke the file as if it were a command.	Permits you to move around in the directory, read the files in the directory.

Finding Out What Permissions Are in Effect

The listing produced by the **ll** command shows the current protection modes for files and directories. Figure 2-5 illustrates an **ll** listing.

- The first column of the listing indicates whether the file is a directory ("d"), a regular file ("-"), a character-type special file ("c"), a block-type special file ("b"), or a multiplexed special file ("m"). (These terms are defined in the *Glossary* of this manual.)
- The next three columns show whether the file's owner has read ("r"), write ("w"), and execute ("x") permission, respectively. A letter indicates that the owner has that type of access permission. A minus sign (-) indicates that the permission has been denied.
- The second three columns list group permissions.
- The final three columns list the permissions for other users on the system.

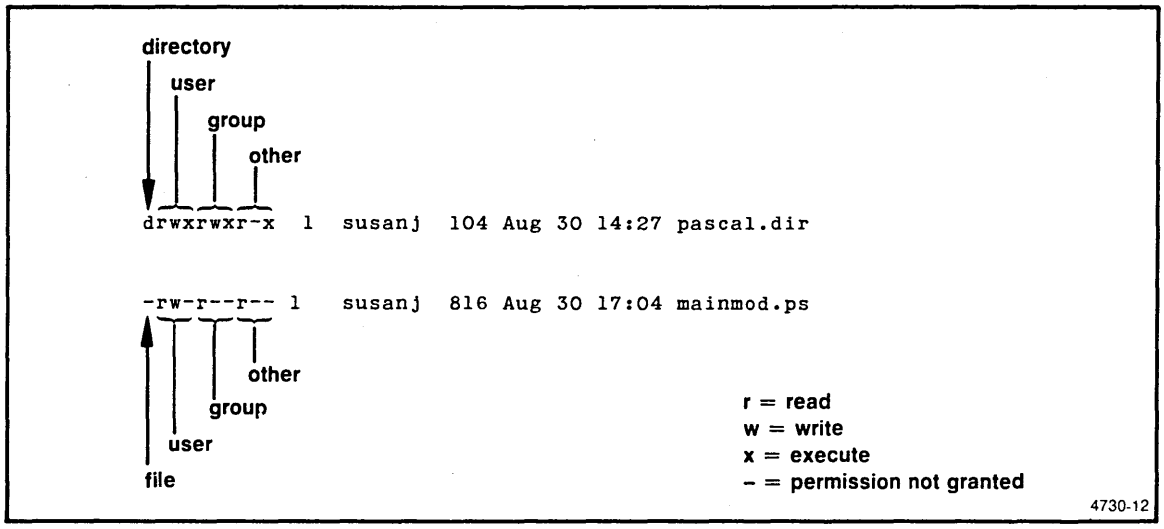


Fig. 2-5. File protection modes.

The output from the **ll** command shows the current protection modes. *Pascal.dir* is a directory that anyone may read, and that the owner and other members of the owner's group may write into. *Mainmod.ps* is a file that can be read by anyone on the system, can be written to only by the owner, and is not executable.

Changing the Protection Modes

The protection modes shown in Fig. 2-5 are the TNIX default modes for a directory and a file. You can use the **chmod** (CHange MODes) command to change protection modes. (Only a file's owner or a superuser—someone logged into the root account—can change a file's protection modes.) For more information, refer to the heading "File Protection" in the *Operating Procedures* section of this manual, or enter **man chmod**.

System File Structure

Figure 2-1, earlier in this section, illustrated the system file tree. Table 2-4 lists the TNIX system directories, which are contained in the top level of the TNIX directory hierarchy. For information on the format of a particular system file, enter the **man** command plus the filename. For example, **man group** displays information about the */etc/group* file.

Table 2-4
TNIX Directory Hierarchy

Directory	Contains	Examples
/bin	Frequently used commands	dsc50, mail, make, sort
/etc	Essential data Superuser commands	group, mount, termcap getty, passwd
/usr /usr/bin	User files Less frequently used commands	johnr, joeb, marya ace
/dev	Peripheral devices[a]	hsi0, tty1
/tmp	Temporary files	
/lib	Auxiliary files, usually TNIX commands; runtime support libraries	

[a] **Each peripheral device (such as a line printer, terminal, or disk drive) is considered a file, and has an entry in the *dev* directory.**

TNIX COMMAND LANGUAGE

The TNIX shell is a program that interprets commands and oversees their execution. The following paragraphs briefly describe features of the shell that enable you to redirect the input and output of commands, connect the output of one command with the input of a second command, and execute commands concurrently. This subsection also describes TNIX command format and control characters. The *Shell Programming* section of this manual contains information on advanced features of the shell, and shows how to use the shell's programming language.

The Shell

When you log in to TNIX, the TNIX command interpreter—the shell—issues a prompt (generally a "$"), and waits for you to enter commands. When you enter a command, the shell acts as a command interpreter: it divides the command line into strings of characters, expands any pattern-matching characters, then executes the specified commands.

TNIX Command Format

A TNIX command consists of one or more "words", separated by blanks. The first word must be the name of an executable file (either a command or an executable command file that you've created). The shell passes the remaining "words" as flags and arguments to the command.

A flag (or option) is a single character preceded by a hyphen; flags may be concatenated. An argument is a valid TNIX filename or string. For example, the following command deletes the contents of the directory *oldfiles*. The "r" and "i" are flags; "oldfiles" is an argument:

```
$ rm -ri oldfiles
```

Flags and arguments are collectively termed "parameters".

Two or more commands may appear on the same line, separated by a semicolon. For example:

```
$ cd eddir ; ls
```

The spaces before and after the semicolon are not required; they are used here for readability.

The shell does not interpret an input line until you press the RETURN key.

Command Input and Output

Ordinarily, the shell sends the output of a command to a file called the "standard output" file, which is generally assigned to your terminal. Similarly, the shell expects input for commands to come from the "standard input" file, also the terminal. However, as Fig. 2-6 shows, the shell can also reassign the input and output of each program, file, command, or device:

- The output from your terminal, from a file, or from a program or command can be connected to a program's input.
- A program's output can be sent to your terminal, to a file, to another program's or command's input, or to any combination of these.
- Error messages from a program or command can be sent to your terminal, to a file, to another program's input, or to any combination of these. The file can be the same file to which the program's output is connected.

The following paragraphs show how you can reassign the standard input and output files to other files and devices, and how you can use "pipes" to connect the output of one command directly with the input to another command.

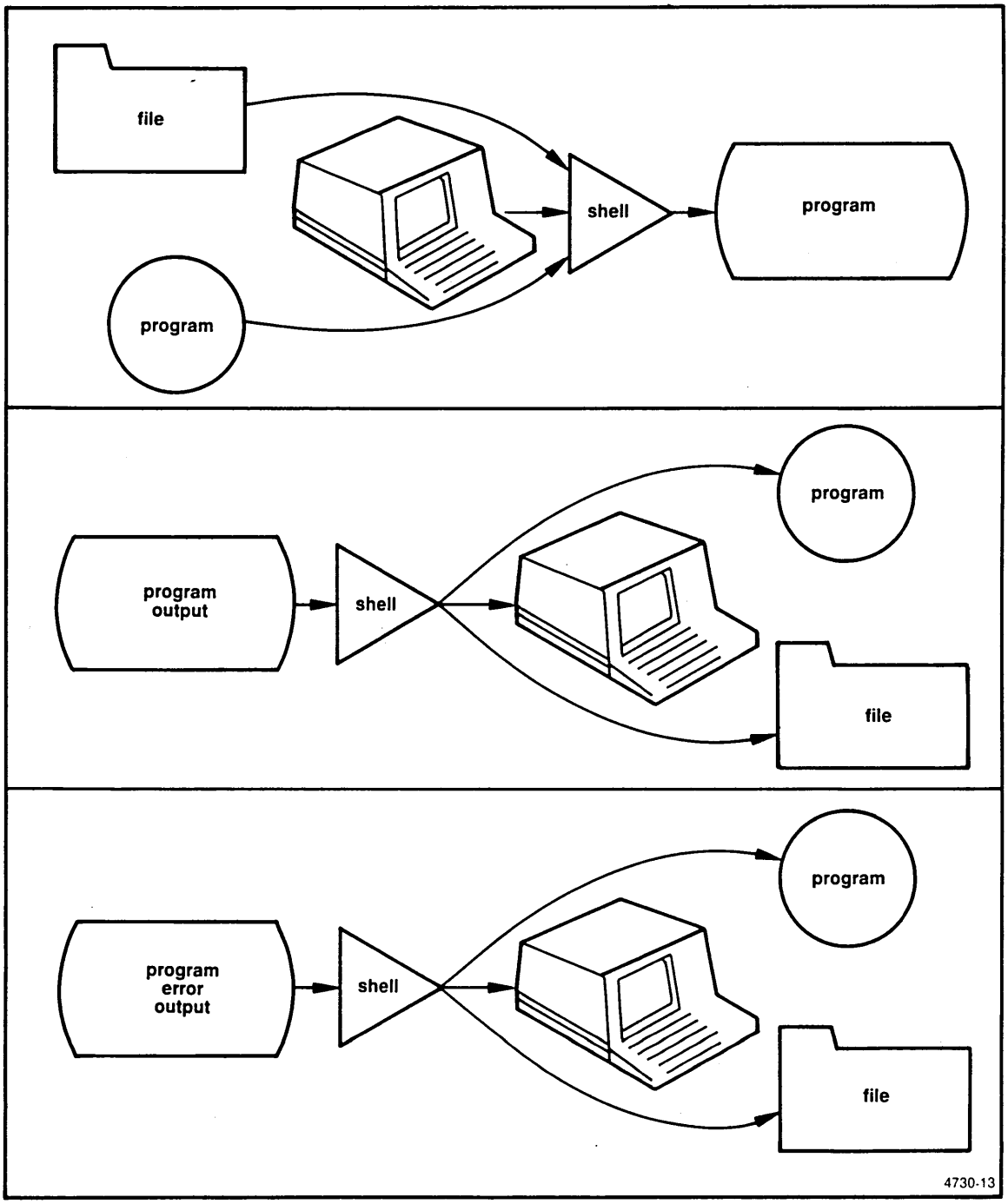


Fig. 2-6. Redirecting input and output.

TNIX commands normally take their input from and send their output to the terminal. However, you can reassign standard input, standard output, and standard error to other files and devices.

Redirecting Output

The ">" (right angle bracket) lets you redirect output to a file other than the standard output file. For example, the following command counts the number of lines in a file and stores the results in a file called *linecount* rather than displaying them on the screen:

```
$ wc -l longfile >linecount
```

If *linecount* does not exist, it is created. If it does exist, its contents are overwritten.

To **append** the results to an existing file, use ">>":

```
$ wc -l longfile >>linecount
```

Redirecting Input

The "<" (left angle bracket) redirects standard input. For example, the **mail** command usually takes its input from standard input: the text you type in after the **mail** command is sent to the specified user. However, you can also use an editor to create a file, and then mail that file. The following example mails a file called *reminder* to user jackf:

```
$ mail jackf <reminder
```

Pipes: Connecting Commands

The pipe operator "¦" connects the output of one command to the input of another, as shown in Fig. 2-7.

The following example uses a pipe to determine the number of users on the system.

```
$ who ¦ wc -l
```

First, the **who** command generates a list of all current users. The list is not displayed on your terminal, however, but is fed into the **wc** command, which (with the **-l** option) counts the number of lines of input from **who**. In order to accomplish this task without using pipes, you'd have to create a temporary file:

```
$ who >temp
$ wc <temp
$ rm temp
```

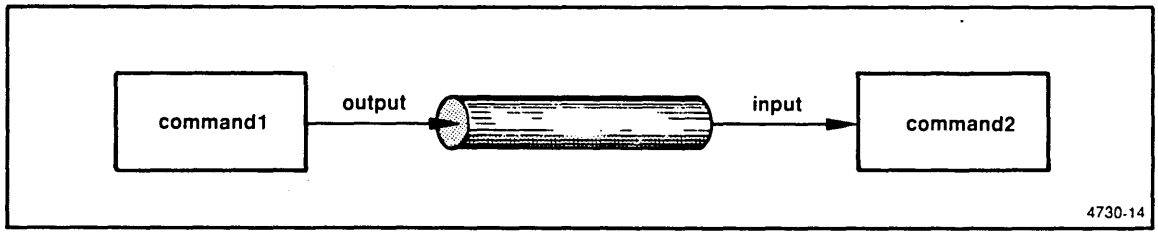


Fig. 2-7. Pipes.

Pipes and I/O redirection can be combined. Thus, the following command saves the results of the previous example in a file:

```
$ who ¦ wc -l  >whocount
```

Pipes can also link any number of commands. The following command prints a sorted list of users who are currently logged on:

```
$ who ¦ sort ¦ lplr
```

Command Execution

The following paragraphs describe how to execute commands concurrently, and how to interrupt one program or command to perform other system tasks.

Background Mode: Executing Commands Concurrently

Ordinarily, when you enter a command, you have to wait until the command finishes executing before entering another command. When you place a command in background mode, however, the shell prompt returns immediately. You can enter additional commands, which will execute concurrently with the original command.

The "&" operator at the end of a command line places the command in background mode. The following example assembles a program in the background:

```
$ asm sub.obj sub.asml sub.asm 2>errors &
1427
$
```

This example also redirects the error output ("2>") to a file called *errors*. (The "2" is the *file descriptor* for the standard error file.) Ordinarily, error messages are displayed on the system terminal ("standard error"). By redirecting the error display to a file, you ensure that error displays won't interrupt whatever you're doing at the terminal.

When a background process begins execution, the system displays a process ID number that identifies the command. (In the previous example, the process ID number is 1427.) You can use this process ID number to monitor the progress of a background job (by entering the **ps** command), or to abort a background job. The following command aborts the assembly example:

```
$ kill 1427
```

Another way to abort a background command is to enter **kill 0**, which terminates all your currently executing commands.

Your background commands are terminated when you log out. However, you can use the **nohup** ("no hangup") command to continue execution of the commands when you log out. For example:

```
$ nohup asm sub.obj sub.asml sub.asm 2>errors &
```

Concurrently Executing a List of Commands. You can also place a list of commands in the background, to be executed concurrently. For example:

```
$ (
> cd
> nroff -ms status.week10 >week10.lpr
> lplr week10.lpr
> echo Status report is printed
> )&
```

This command performs the following actions:

1. temporarily changes the current directory (until the ")" is reached) to the HOME directory;
2. formats a status report with the optional **nroff** text formatter;
3. prints the formatted status report; and
4. displays a message on your terminal telling you that the status report has been formatted and printed.

While the document is formatting and printing, you can continue to enter commands.

Multitasking

TNIX provides several ways to suspend what you're doing and start another system activity. The following paragraphs describe two examples.

Escape from an Editor. You can temporarily escape to the shell from an editor without terminating the edit session. Table 2-5 shows how to escape to the shell from the standard TNIX editor (ed), from the optional TEKTRONIX ACE Screen Editor (ACE), and from the optional TEKTRONIX Language-Directed Editor (LDE).

Table 2-5
Escape to Shell from an Editor

Editor	Escape for One Command	Escape for Several Commands
ed, ACE	!command<CR>	!sh commands CTRL-D
LDE	command<CTRL-X> [a]	sh <CTRL-X> [a] commands CTRL-D

[a] For terminals other than the TEKTRONIX 4105M terminal, use the key(s) configured for Execute System Command.

Switch Users. The **su** (switch user) command lets you temporarily use someone else's terminal while he or she is logged on. You start by entering **su** and your login name:

```
$ su yourname
password:                    [enter your password]
$
.                            [do the work you need to do]
.
<CTRL-D>                     [then return the terminal to the original user]
```

After you've entered the **su** command line and supplied your password, the system recognizes you as the current user, but does not change the current directory or user name, and does not execute your *.profile* file (which is executed when you log in). If you have redefined the $ prompt, for example, you will see the "$" rather than your own prompt string.

The CTRL-D returns the terminal to the original user.

Control Characters

Table 2-6 lists the TNIX control characters that you are likely to use in your interactions with the shell.

Table 2-6
Control Characters

Character	How Used
CTRL-C	Interrupt command or program execution
CTRL-D	Terminate terminal input; terminate the current shell [a]
CTRL-H	Backspace: delete input character
CTRL-K	Retype successive characters of the previous line
CTRL-Q	Resume output display
CTRL-R	Reprint input line
CTRL-S	Suspend output display
CTRL-U	Delete input line

[a] **If the current shell is not a subshell, you will receive a message telling you to type "logout" to log out.**

CUSTOMIZING YOUR TNIX ENVIRONMENT

There are several ways in which you can customize TNIX to suit your programming needs:

- You can set "environment variables" and place them in a *.profile* file where they will be set automatically each time you log in.
- You can write your own commands (by using the shell programming language or by combining existing commands into an executable file).

Environment Variables

Environment variables provide information to the shell and to certain commands. The *Shell Programming* section of this manual presents detailed information about environment variables and other shell variables. The following paragraphs discuss the most frequently used environment variables and tell how to set them.

HOME Specifies the default directory for the **cd** command. If you have not assigned a value to HOME, its value is your login directory, */usr/yourname*.

IU An integer in the range 0-7 which specifies the 8560 HSI I/O Port to which you have connected an 8540 Integration Unit or 8550 Microcomputer Development Lab.

PATH Specifies a list of directories (each preceded by a colon) that the shell will search for commands, and the order in which it will search them. If PATH is not set, the shell searches the current directory, then */bin* and */usr/bin*.

TERM The name by which your terminal type is known to TNIX. Terminal type names are listed in the */etc/termcap* file.

uP The name by which a target microprocessor is known to TNIX. The uP variable must be set before you use any 8560 compiler or assembler.

PS1 The primary shell prompt string. Defaults to "$".

To set an environment variable, you first assign it a value, then use the **export** command to make the variable available to all subshells. Here are some examples:

```
$ HOME=/usr/jones/mainprogram/specs; export HOME

$ PATH=:/usr/smith/bin:/bin:/usr/bin; export PATH

$ TERM=4105; export TERM

$ uP=68000; export uP

$ PS1="command: "; export PS1
```

You can also assign values to several variables, then export all of them at once:

```
$ TERM=4105; uP=68000; export TERM uP
```

When you use an environment variable in a command line, you must preface it with a "$". For example:

```
$ cd $HOME/file1
```

You can use the **echo** command to display the value of an environment variable. For instance:

```
$ echo $TERM
4105
```

The **set** command displays the values of all your environment variables.

The .Profile File

When you set an environment variable from the shell, the definition is valid only for the current login session. However, if you place the definition in your *.profile* file, it will become part of your TNIX environment whenever you log in.

The *.profile* file is a file in your HOME directory that is executed each time you log in. Because the filename is preceded by a period, the file is not listed by most forms of the **ls** command. However, the file **is** listed by **ls -a**, and can be modified using any 8560 text editor. For an example of how to modify your *.profile*, see the procedure "Invoking Commands Automatically Upon Login" in the *Operating Procedures* section of this manual.

Creating Your Own Commands

Each TNIX command is an executable file. The following paragraphs show how you can create your own commands by placing one or more commands in a file and then executing the file. This discussion also describes how to create a personal programs directory to contain any commands you create, and how to share your commands with other users.

To take an example, let's say that you frequently assemble the same file. To avoid having to repeatedly type the full assembler invocation, you decide to place the assembler invocation line in an executable file called *myasm*. You can then invoke the assembler simply by executing your file.

Create the File

Using any 8560 editor, create a file named *myasm* and place the **asm** command line in it:

```
$ asm sub.obj sub.asm1 sub.asm 2>errors &
```

Execute the File

There are several ways to execute the file:

1. Enter a period followed by the filename:

   ```
   $ . myasm
   ```

   You must use this method if your command will change your current "environment"—for example, if it will change the current directory or initialize or alter any environment variables.

2. Use the **sh** command:

   ```
   $ sh myasm
   ```

   This form of execution is especially useful when you want to pass parameters to the command. For example, the following command executes a file called **run** with the shell's **-x** execution trace option set:

   ```
   $ sh -x run
   ```

   (The *Shell Programming* section of this manual contains further information on how to pass parameters to a command or program.)

3. Change the protection modes to make the file executable, then execute the file by typing the filename:

   ```
   $ chmod u+x myasm
   $ myasm
   ```

Create a Personal Programs Directory

TNIX stores commands in the system's *bin* and */usr/bin* directories. It's a good idea to create a directory called *bin* in your HOME directory and place any commands you create in that directory. You will also need to tell the shell to check this directory when it looks for command names. To do this, add the following lines to your *.profile* file:

```
PATH=":/usr/yourname/bin$PATH"
export PATH
```

These two lines tell the shell to look for commands first in the current directory, then in your *bin* directory, then in the standard TNIX command directories (*/bin* and */usr/bin*).

Sharing Commands

You can also make your command available to other users on the system. First, make your file executable by other users:

```
$ chmod o+x myasm
```

Other users can now execute the file by typing its full pathname (*/usr/yourname/bin/myasm*) or by linking to the command. You can also place the command in the system's */usr/bin* directory. This enables other users to execute the command by simply typing the command name.

You must have **superuser status** to place the command in the */usr/bin* directory. If you do not have superuser status, ask your system manager to put the command in */usr/bin*.

SUMMARY

This section has provided an overview of the TNIX operating system and shown you the basic steps involved in creating your own commands. Continue on to the next two sections for more information about these topics.

- Section 3, *Operating Procedures*, shows the commands that you enter to perform many common system tasks.
- Section 4, *Shell Programming*, describes the shell programming language.

Section 3
OPERATING PROCEDURES

Page

Section 3

OPERATING PROCEDURES

INTRODUCTION

This section presents some procedures to help you use your 8560. Most of the concepts and terms required to perform these procedures were discussed in the previous section of this manual, *TNIX Operating System.* If you encounter unfamiliar terms as you go through the procedures, refer to the *Glossary* of this manual.

Unless otherwise noted, you can perform these procedures while using Keyshell. All procedures can be performed through the shell.

The procedures in this section are presented in the following format:

Description:
A summary of the action(s) performed by the procedure.

Procedure:
The information entered or displayed at the system terminal. Words appearing in **bold** are parameters that you select. Words within brackets [ ] are for your information only; do not enter them. Remember to press the RETURN key after each command.

Parameters:
A description of the values you supply.

Comments:
Additional information.

Example:
One or more demonstrations of correct entry format.

See also:
References to related procedures, manuals, and online information.

GETTING STARTED

Powering Up the 8560

Description:

This procedure describes how to power up the 8560. Your 8560 system must be configured as described in the *8560 Series System Manager's Guide*.

Procedure:

1. Power up the system console and any other peripherals.
2. Turn on the AC power switch on the 8560 back panel, then turn on the DC power switch on the front panel. The TNIX operating system boots automatically from the hard disk. The flexible disk drive must be open.
3. TNIX displays a welcome message and asks whether you want it to check the file system. Type **y**. The check takes several minutes.
4. TNIX asks you to enter the current date. Your answer should include the time, for example, **30-aug-83 9:20**.
5. TNIX asks if you want to remain "single user". Type **n** unless you are performing system maintenance activities.
6. TNIX displays the login prompt.

See also:

- Logging In

Logging In

Description:

This procedure logs you in to the TNIX operating system.

Procedure:

login: **username**
password: **password** [not echoed]

Parameters:

username—The sequence of characters that defines your user account on the system. The user name (or "login name") allows the system to distinguish you from other users. User names are public information.

password—The sequence of ASCII characters you must enter to establish that you are authorized to use the account. Passwords are non-public information: the system stores only an encrypted copy.

Comments:

If your account has no password, none will be requested. If you do not yet have a login name, talk to your system manager about getting an account.

If you have just turned on or connected your terminal and your first login attempt fails, try again.

You can log in from another account by typing **login**; the old account will be automatically logged out, and you will be prompted for your password.

Logging In Through an 8540

Description:

This procedure shows you how to log in to the 8560 through an 8540. The 8560 must be powered up and running the TNIX operating system. Refer to the *8560 Series System Manager's Guide* for information on how to configure your 8560 and 8540.

Procedure:

[Boot up the 8540.]

[If the terminal does not display a prompt, the 8540 has probably booted up in TERM mode.]

[If the terminal displays a > prompt, the 8540 has booted up in LOCAL mode. Enter the following command line to enter TERM mode:]

```
> config term
```

[Now log in to TNIX.]

Logging In Through an 8550

Description:

This procedure logs you into the 8560 from an 8550. The 8560 must be powered up and running the TNIX operating system. The 8550 and 8560 communicate via a line that runs from the DTE jack (J101) on the 8301 to an HSI I/O jack on the 8560. The HSI I/O jack must be jumpered for RS-232-C communication.

Procedure:

```
> config term t=7
```

[From another terminal, enter the following command:]

```
$ stty IU >/dev/ttyn
```

[You can now log in to TNIX on the terminal that is connected to the 8550.]

Parameters:

n—A number between 0 and 7 (inclusive) which specifies the 8560 HSI I/O port that the 8550 is connected to (on the 8560's rear panel).

Comments:

If your system manager has modified the system's */etc/ttys* file to include a line such as "1cttyn" for your port number, you can establish communication with the 8560 by simply typing the **config term t=7** command. Refer to the *8560 Series System Manager's Guide* for information on how to modify the */etc/ttys* file.

Selecting the 8540 or 8550

Description:

This procedure selects an attached 8540 or 8550 when your terminal is connected directly to the 8560 and not to an 8540 or 8550.

Procedure:

$ IU=**n**; export IU

Parameters:

n—A number between 0 and 7 (inclusive) which specifies the 8560 HSI I/O port that connects to the 8540 or 8550.

Example:

```
$ IU=3; export IU
```

This example tells the 8560 that I/O port number 3 is connected to either an 8540 or 8550. The terminal you are using can now issue commands to the 8540 or 8550 as if it were connected to the 8540 or the 8550.

Invoking Commands Automatically Upon Login

Description:

This procedure shows how to modify your *.profile* file, a file of commands that is automatically executed each time you log in.

Procedure:

[Edit the *.profile* file (in your login directory) and add any desired commands to it.]

Example:

Assume that you've added the following commands to your *.profile* file:

```
PS1="yes? "
echo Number of users is: `who | wc -l`
PATH=:$HOME/.bin:/bin:/usr/bin
export PS1, PATH
```

This example performs several functions each time you log in.

The first line sets your prompt string (PS1) to **yes?**.

The second line tells you how many users are currently logged in (including yourself).

The third line sets the PATH environment variable, which tells the system where to search for command names. TNIX will first search in a directory called */.bin* in your home directory, then in the system's */bin* directory, and finally in the system's */usr/bin* directory. (TNIX commands are stored in */bin* and */usr/bin*.) Only after searching these three directories will TNIX respond that a command was not found. This allows you to have a private set of commands (executable files) in your home directory.

The final line makes the definitions of PS1 and PATH available to the login shell and all subshells.

See also:

- Re-initializing Your Default .profile
- Section 2 of this manual, *TNIX Operating System*

Re-initializing Your Default .profile

Description:

If you accidentally garbled your *.profile* file, you can use this procedure to restore the file to its original state.

Procedure:

$ cp /usr/lib/default.profile /usr/**yourname**/.profile

Parameters:

Yourname—Your login name.

*This procedure overwrites your existing **.profile** file.*

Changing Your Password

Description:

This procedure changes your password.

Procedure:

$ passwd

[The system asks for your current password, then for the new password.]

Comments:

A password longer than six characters is more secure than a shorter password. If you enter a password of less than six characters, the **passwd** command will ask for a longer one. If you continue to respond with a short password, it will be accepted.

Example:

```
$ passwd
Changing password for johnd
Old password: friedegg  (not echoed)
New password: scramble  (not echoed)
Retype new password: scramble  (not echoed)
```

This example changes johnd's password from **friedegg** to **scramble**.

Selecting a Target Processor

Description:

This procedure selects the target microprocessor for emulation on the 8540 or 8550, and identifies the selected processor on the 8560 by assigning a value to the TNIX variable **uP**.

Procedure:

$ sel **target**
$ uP=**target**; export uP

Parameters:

target—The name of the target microprocessor. The name may differ slightly from the more common name of the microprocessor. For more information, see the *Emulator Specifics* manual for your emulator.

Comments:

The first command (**sel**) is necessary only if you are communicating through an 8540 or 8550 to the 8560. The second command (setting and exporting the uP variable) is necessary only if your 8560 has chip-specific software.

You may want to add these commands to your *.profile* file. That way, they will be executed each time you log in.

Example:

```
$ sel 68000
$ uP=68000; export uP
```

This example selects the 68000 microprocessor.

See also:

- Invoking Commands Automatically Upon Login

Changing Your Terminal's Baud Rate

Description:

This procedure changes the terminal's baud rate. This procedure has no effect if your terminal is connected to an 8540 or 8550.

Procedure:

$ stty **baudrate**
[Now, set your terminal to this baud rate.]

Parameters:

baudrate—The desired baud rate for your terminal: 300, 600, 1200, 2400, 4800, or 9600.

Comments:

The **stty** command tells the system that your terminal will communicate at a particular baud rate. The system will now expect to receive that baud rate from your terminal. Therefore, before you do anything else, you must reset the baud rate on your terminal. Refer to the manufacturer's manual for your terminal for details.

Logging Out

Description:

This procedure allows you to exit from the current shell and to terminate contact with the system.

Procedure:

$ logout

Comments:

The **logout** command terminates the current shell. If that shell was a subshell, you will still be logged in. If it was the login shell, you are logged out, and the system will respond with a **login** prompt.

Logging Out Through an 8540 or an 8550

Description:

This procedure logs out from the 8560 and terminates communication between the 8560 and the 8540 or 8550 to which your terminal is connected.

Procedure:

$ config local; logout

Powering Down the 8560

Description:

This procedure describes how to power down the 8560. You must be a superuser (logged into the **root** account) to perform this procedure.

Procedure:

[First, remove the flexible disks from any 8550s connected to the 8560. Turn off all power to the 8540s and 8550s connected to the 8560. Make sure all users are logged off the 8560.]

shutdown

[TNIX asks you to wait while it shuts itself down. When TNIX gives you permission, remove the flexible disk from the 8560 (if there is one in the drive). Then turn off the DC power switch on the 8560 front panel and the AC power switch on the 8560 back panel.]

*Be sure to issue the **shutdown** command before you power down the system. If, for any reason, you cannot issue a **shutdown** command, you must issue a **sync** command while logged in to the **root** account, and kill any user processes. Failure to issue the shutdown or **sync** commands could scramble the file system.*

DIRECTORY MANIPULATION

Creating a Directory

Description:

This procedure creates a new directory.

Procedure:

$ mkdir **directory**

Parameters:

directory—The pathname of the directory to be created.

Comments:

An empty directory is created at a location in the file tree specified by the pathname.

Example:

```
$ mkdir /usr/jpgetty/income
```

This example creates the directory *income* in the directory *jpgetty*.

Displaying the Name of the Current Directory

Description:

This procedure displays the pathname of the current directory.

Procedure:

$ pwd

Example:

```
$ pwd
/usr/francisd/goldenhind/world
```

This example tells you that your current directory is *world*. The entire pathname of the directory is displayed, to distinguish it from any other directory you may have called **world**.

Displaying the Contents of a Directory

Description:

This procedure lists the files in the specified directory.

Procedure:

$ ls **directory**

Parameters:

directory—The pathname of the directory you want to list. If you do not specify a directory, the contents of the current directory are listed.

Comments:

The **ls** command has a number of options that allow you to specify what information will be listed. For an explanation of these options, enter the command **man ls**.

Example:

```
$ ls
cycle.asm
cycle.obj
d45.asm
restart.asm
restart.back
restart.dxfv
spin.asm
u3.asm
```

This example lists the files in the current directory.

Moving to Another Directory

Description:

This procedure moves you to another directory, which becomes the current directory.

Procedure:

$ cd **directory**

Parameters:

directory—The pathname of the directory you want to move to. If you do not specify a directory, **cd** moves you to your home directory (/usr/yourname).

Example:

```
$ cd ..
```

This example moves you to the current directory's parent directory.

```
$ cd /usr/smith/programs/games
```

This example moves you to a directory named *games*, which is a subdirectory of Smith's *programs* directory.

See also:

- Section 2 of this manual, *TNIX Operating System*

Deleting an Empty Directory

Description:

This procedure removes an empty directory.

Procedure:

$ rmdir **directory**

Parameters:

directory—The pathname of the directory to be deleted.

Comments:

If the directory contains any files or subdirectories, the message "directoryname not empty" is displayed.

Examples:

```
$ rmdir emptydir
```

See also:

- Deleting All Files from a Directory

Deleting a Directory and the Files Within It

Description:

This procedure removes a file tree and all its subtrees, querying you before each deletion.

Procedure:

$ rm -ir **directory**

Parameters:

directory—The pathname of the directory that you want to remove.

Comments:

A file is destroyed only if you remove (unlink) the file from the only directory that it resides in.

If you omit the **-i** flag, TNIX removes the files without querying you.

Example:

```
$ rm -ir files
files/file1: y
files/file2: y
files/file3: y
files/morefiles: y
files/morefiles/file1: y
files/morefiles/file4: y
files/morefiles/file6: y
files/morefiles: y
files: y
```

This example removes the directory *files* and its contents. (Note that there are two distinct *file1* files in two distinct directories.)

The system queries you before it removes each file from the directory. Type **y** to remove the file. Any response that begins with a character other than **y** will not remove the file.

The first time the system queries *files/morefiles:*, it is asking you whether to search the directory *morefiles*. After you have removed *file1*, *file4*, and *file6* within the *morefiles* directory, the system asks you whether or not to remove the directory itself. You may remove the directory only if it is empty.

Duplicating a Directory

Description:

This procedure creates a duplicate copy of a directory and all its subdirectories and files (i.e., an entire file tree).

Procedure:

$ cp **directory1 directory2**

Parameters:

directory1—The pathname of the directory to be copied.

directory2—The pathname of the directory that will be the copy.

Comments:

If *directory2* does not exist, it is created and will contain a copy of the contents of *directory1*. If *directory2* does exist, the copy of *directory1* becomes a subtree of *directory2*.

Directory2 must not be contained within the *directory1* subtree.

Example:

```
$ cp oldtree newtree
```

FILE MANIPULATION

Creating a File

Description:

This procedure uses the TNIX Editor to create a new file.

Procedure:

$ ed **file**
?file

a	**[the ed command to append text]**

[Enter text.]

w	**[save the file]**
nnnn	**[ed displays the number of characters in the file when it writes to the disk.]**
q	**[quit]**

Parameters:

file—The pathname of the file to be created.

Comments:

Tektronix offers two optional, screen-oriented editors for use on the 8560: the ACE Screen Editor and the Language-Directed Editor.

Example:

```
$ ed newfile
?newfile
a
Yesterday upon the stair
I met a man who wasn't there.
.
w
53
q
```

This example creates the file *newfile* and appends two lines to the empty file.

See also:

- Section 5 of this manual, *TNIX Editor*

Renaming or Moving a File

Description:

This procedure moves or renames a file or directory.

Procedure:

$ mv **pathname1 pathname2**

Parameters:

pathname1—The file to be moved or renamed.
pathname2—The new name or location of the file.

Example:

```
$ mv /usr/joe/directory1/oldname /usr/joe/directory2/newname
```

The file *oldname* is moved from *directory1* to *directory2*, and is renamed *newname*.

Duplicating a File

Description:

This procedure creates a duplicate copy of a file.

Procedure:

$ cp **pathname1 pathname2**

Parameters:

pathname1—The original file.
pathname2—The new copy of the file.

Comments:
If the new file *pathname2* already exists, it will be destroyed.

Example:

```
$ cp /usr/mikem/directoryl/origfile copyfile
```

This example copies the file *origfile* from *directory1* into the current directory. The new file is named *copyfile*.

Creating a Link to a File

Description:
This procedure creates a new path to an existing file. The name of the file in the new directory does not have to be the same as in the old directory.

Procedure:
$ ln *pathname1 pathname2*

Parameters:
pathname1—The pathname of an existing file.
pathname2—The pathname that represents a path to the file through a second directory.

- If *pathname2* is a simple filename, the file referred to by *pathname1* is linked to an entry called *pathname2* in the current directory.
- If *pathname2* is a full pathname, the file will be placed in the appropriate directory.
- If *pathname2* is not specified, the file will be entered in the current directory with the same name as the last part of *pathname1*.

Comments:
Creating a link to a file is often preferable to copying a file: it saves disk space, and updating one version of the file updates all "copies" of that file. Links are often made to executable files. Links cannot be made across filesystems.

Example:

```
$ ln /usr/georgew/auto/steerage/linkage
```

This example creates a file called *linkage* in the current directory. This file is linked to the file */usr/george/auto/steerage/linkage*.

```
$ ln /usr/barrym/trajectory/resistance friction
```

This example creates a link between a file called *friction* in the current directory and barrym's file *resistance*.

Deleting a File

Description:

This procedure removes a file from a directory.

Procedure:

$ rm **file**

Parameters:

file—The pathname(s) of the file or files to be removed.

Comments:

The file is destroyed only if you remove (unlink) the file from the only directory that it resides in. To determine the number of links to a file, enter **ls -l filename**.

Example:

```
$ rm /usr/emersonj/trash
```

This example removes the file *trash* from emersonj's directory.

```
$ rm myfile yourfile ourfile
```

This example removes three files from the current directory.

```
$ rm *file
```

This example removes from the current directory all files that end with the word "file" (for example, myfile, yourfile, ourfile).

Deleting All Files from a Directory

Description:

This procedure deletes all files from the specified directory. The directory itself is not removed. You are not queried before the files are removed.

Procedure:

$ cd **directory**
$ rm ***

Parameters:

directory—The directory containing the files you want to remove.

Comments:

A file is destroyed only if you remove (unlink) the file from the only directory that it resides in.

See also:

- Deleting an Empty Directory

Concatenating Two or More Files

Description:

This procedure concatenates two or more files (i.e., merges them end-to-end).

Procedure:

$ cat *infile1 infileN ... >outfile*

Parameters:

infile1—The pathname of the first file to concatenate.

infileN—The pathname of the next file to concatenate.

outfile—The pathname of the file created by the concatenation.

*If you attempt to concatenate a file to itself (for example, **cat file.ps >file.ps**), the file will be destroyed.*

Example:

```
$ cat startfile middlefile endfile >wholething
```

This example merges three files and places the results in a new file, *wholething*.

```
$ cat morestuff >>oldfile
```

This example appends (>>) the contents of *morestuff* to the file *oldfile*.

Counting the Lines of a File

Description:

This procedure counts the number of lines in a file.

Procedure:

$ wc -l **file**

Parameters:

file—The pathname of the file whose lines you want to count.

Comments:

The **-l** flag of the **wc** command counts the number of lines in a file; other forms of **wc** count the number of characters and words in a file. For more information, enter **man wc**.

Example:

```
$ wc -l prog1 prog2 prog3
     742 prog1
      38 prog2
     519 prog3
    1299 total
```

Searching for a Specific File

Description:

This procedure searches through a specified file tree for a file, and displays the pathname of the file.

Procedure:

$ find **directory** -name **filename** -print

Parameters:

directory—The directory to be searched.

filename—The file or directory that you want to find.

Example:

```
$ find . -name pilot* -print
/usr/drewl/mktg/pilot.demo
/usr/drewl/traffic/pilot.air
```

This example searches the current directory (".") and its subdirectories and displays the pathname(s) of each file that starts with *pilot*.

Searching for a Pattern in a File

Description:

This procedure searches for a pattern in a file and prints the lines that contain the pattern.

Procedure:

$ grep -ny '**pattern**' **file**

Parameters:

pattern—A pattern as defined for the TNIX Editor. Within the pattern, the following characters have special meanings to the **grep** command:

```
^ $ . [ ] *
```

Because many characters have special meaning to the TNIX shell, it's a good idea to enclose the pattern in single quotes to prevent the shell from interpreting it.

file—The pathname of the file you want to search for the pattern.

Comments:

The **-n** argument tells **grep** to display the line number at which the pattern was found. The **-y** argument specifies that the pattern may include uppercase letters.

Lines longer than 256 characters are truncated.

Example:

```
$ grep -yn oddresult numcrunch.ps
```

This example searches for the pattern **oddresult** in the file **numcrunch.ps**.

See also:

- Section 5 of this manual, *TNIX Editor*

Performing the Same Operation for Several Files

Description:

This procedure performs the same operation repeatedly on a number of files. This procedure uses a multi-line shell command and thus should not be entered while you're using Keyshell.

Procedure:

$ for i in **file1 file2 ...**
> do
> **$i-command-line**
> done
$

Parameters:

file1, file2—The pathnames of the files that the command will affect.

**$i-command-line**—A command line that is ordinary in every respect, except that the filename or pathname in the line is "$i".

Comments:

To perform this procedure while you're using Keyshell, enter the **sh** command as the first line of the procedure, and type CTRL-D on a separate line when the procedure is finished.

Example:

```
$ for i in dir1/*
> do
> ln $i dir2
> done
$
```

This example links all files in *dir1* to *dir2*.

See also:

- Section 4 of this manual, *Shell Programming*

Identifying and Removing Unused Files

Description:

This procedure identifies files that have not been accessed recently, then lets you remove them.

Procedure:

$ ls -altur **directory**

.
.
.

$ rm **file file ...**

Parameters:

directory—The pathname of the directory that you want to examine.
file—The pathname(s) of the file(s) you want to remove.

Comments:

The sorted list of files begins with the least recently accessed file, and proceeds to the most recently accessed.

The file is destroyed only if you remove (unlink) the file from the only directory that it resides in.

See also:

- For an explanation of the flags for the **ls** command, enter **man ls**.

Example:

```
$ ls -altur
total 38
-rw------- 1 bethb      3107 Feb  2 15:08 file3
-rw------- 1 bethb       482 Jul  9 10:40 file2
-rw------- 1 bethb      3107 Jul 10 10:41 f3
-rw------- 1 bethb      2465 Jul 10 10:41 f6
-rw------- 1 bethb      1501 Jul 30 15:57 f0
-rw------- 1 bethb       964 Aug  8 14:03 f4
-rw------- 1 bethb       196 Nov  9 08:17 file1
-rw------- 1 bethb      4608 Nov  9 10:47 f5
-rw------- 1 bethb      1501 Nov 19 10:48 file0
$ rm file3 f3 f6 f4
$ ls -altur
total 17
-rw------- 1 bethb       482 Jul  9 10:40 file2
-rw------- 1 bethb      1501 Jul 30 15:57 f0
-rw------- 1 bethb       196 Nov  9 08:17 file1
-rw------- 1 bethb      4608 Nov  9 10:47 f5
-rw------- 1 bethb      1501 Nov 19 10:48 file0
```

This example examines the current directory and deletes files that have not been recently accessed. Even though *file2* and *f0* were not accessed recently, they were important enough to keep, so they were not deleted.

PRINTING AND DISPLAYING FILES

Displaying a File On Your Terminal

Description:

This procedure displays a file on your terminal.

Procedure:

$ cat **file**

Parameters:

file—The pathname of the file to be displayed.

Comments:

To temporarily stop the display, type CTRL-S. To restart the display, type CTRL-Q. To kill the display, type CTRL-C.

Displaying a File a Screenful at a Time

Description:

This procedure displays a file on your terminal a screenful at a time. Press the space bar to advance to the next screenful.

Procedure:

$ more **file**

Parameters:

file—The pathname of the file to be displayed.

Displaying a File with Line Numbers

Description:

This procedure displays a file on your terminal with line numbers.

Procedure:

$ pr -nt **file**

Parameters:

file—The pathname of the file to be displayed.

Example:

```
$ pr -nt /usr/carls/cosmos
```

This example displays the file *cosmos*, with line numbers, on your terminal.

Printing a File

Description:

This procedure prints a file on the line printer.

Procedure:

$ lpNr **file**

Parameters:

N—The line printer number.
file—The pathname of the file to be printed on the line printer.

Example:

```
$ lplr bl.asm
```

This example prints the file *b1.asm* on line printer 1.

```
$ pr file1 file2 ¦ lp2r
```

This example paginates the two files and sends the results through a pipe (¦) to line printer 2.

Printing a File with Line Numbers

Description:

This procedure prints a file with line numbers.

Procedure:

$ pr -n **file** ¦ lpNr

Parameters:

file—The pathname of the file to be printed on the lineprinter.
N—The line printer number.

Example:

```
$ pr -n /usr/fredericz/words ¦ lplr
```

This example prints the file *words*, with line numbers, on line printer 1.

Checking the Print Queue

Description:

This procedure lists the files that are waiting to be printed.

Procedure:

$ ls -l /usr/spool/lp*

Example:

```
$ ls -l /usr/spool/lp*

/usr/spool/lpl:
total 0

/usr/spool/lp2:
total 83
-rw-r--r-- 1 barneyr      58 Jun 12 07:45 dfA15931
-rw-rw-rw- 2 barneyr   29792 Jun 10 19:01 lfA15931
-rwsr-xr-x 2 daemon    11536 May 14 14:54 lock
```

The print queue contains barneyr's file, as well as an initial banner. The "daemon" is a process that executes independently and performs system maintenance tasks.

Removing a File from the Print Queue

Description:

This procedure shows how to remove a file from the print queue.

Procedure:

$ ls -l /usr/spool/lp*

.

.

$ rm /usr/spool/lp**N/fileid**

Parameters:

N—The number of the line printer queue containing the file that you want to remove.

fileid—The identification number of the file to be removed.

Example:

```
$ ls -l /usr/spool/lp*

/usr/spool/lpl:
total 0

/usr/spool/lp2:
total 83
-rw-r--r-- 1 billk        58 Jun 12 07:45 dfA15931
-rw-rw-rw- 2 billk     29792 Jun 10 19:01 lfA15931
-rwsr-xr-x 2 daemon    11536 May 14 14:54 lock

$ rm /usr/spool/lp2/*A15931
```

This example removes from the print queue the file (lfA15931) that was submitted by user billk. The preceding header file (dfA15931) is also removed.

FILE PROTECTION

Protecting a File from Other Users

Description:

This procedure protects a file from access by all other users.

Procedure:

$ chmod go-rwx **pathname**

Parameters:

pathname—The file or directory whose protection status you are changing.

Comments:

Only the owner of a file or the superuser may change its protection status.

For an explanation of the TNIX protection modes, refer to the *TNIX Operating System* section of this manual.

Example:

```
$ ls -l personnel
-rwxrwxrwx 1 tomp        127 Oct 11 16:47 personnel
$ chmod go-rwx personnel
$ ls -l personnel
-rwx------ 1 tomp        127 Oct 11 16:47 personnel
```

This example prevents other users from reading, writing to, or executing tomp's *personnel* file. The **ls** command verifies that the protection status was changed correctly.

See also:

- Online information: enter **man chmod**

Write-Protecting a File from Other Users

Description:

This procedure protects a file from being written to by any other users.

Procedure:

$ chmod go-w **pathname**

Parameters:

pathname—The file or directory whose protection status you are changing.

Comments:

Only the owner of a file or the superuser may change its protection status.

For an explanation of the TNIX protection modes, refer to the *TNIX Operating System* section of this manual.

Example:

```
$ ls -l mb50file
-rwxrwxrwx 1 maryr       273 Dec 14 14:21 mb50file
$ chmod go-w mb50file
$ ls -l mb50file
-rwxr-xr-x 1 maryr       273 Dec 14 14:21 mb50file
```

This example removes write permission from everyone but the user (owner) of the file, maryr.

See also:

- Online information: enter **man chmod**

Adding Read and Execute Permission to Other Users

Description:

This procedure enables all other users to read and execute a file.

Procedure:

$ chmod go+rx **pathname**

Parameters:

pathname—The file or directory whose protection status you are changing.

Comments:

Only the owner of a file or the superuser may change its protection status.

For an explanation of the TNIX protection modes, refer to the *TNIX Operating System* section of this manual.

Example:

```
$ ls -l teamwork
-rwx------ 1 johnf       461 Apr 21 13:21 teamwork
chmod go+rx teamwork
ls -l teamwork
-rwxr-xr-x 1 johnf       461 Apr 21 13:21 teamwork
```

This example adds read and execute permission to everyone but the user (owner) of the file, johnf, who already has full permission.

See also:

- Online information: enter **man chmod**

STATUS INFORMATION

Determining Who Is On the System

Description:

This procedure displays a list of who is currently logged in, what terminal each user is on, and when each user logged in.

Procedure:

$ who

Example:

```
$ who
happy    tty1   Jul  1 09:04
doc      tty2   Jul  1 04:56
dopey    tty3   Jul  1 08:51
grumpy   tty5   Jul  1 08:14
```

See also:

- Online information: enter **man ps**

Determining Who is Logged in on a Terminal

Description:

This procedure shows who is logged in on the terminal at which the command is entered. The terminal number and the time the user logged in are also shown.

Procedure:

$ who am i

Example:

```
$ who am i
corleaw    tty5   Aug  5 15:27
```

Determining the Date and Time

Description:

This procedure displays the current date and time.

Procedure:

$ date

Example:

```
$ date
Mon Apr 20 14:42:21 PDT 1981
```

Determining What the System Is Doing

Description:

This procedure displays a status line for each TNIX command that is currently executing or waiting for execution.

Procedure:

$ ps -ax

Comments:

The state of the system may undergo changes even as the **ps** command is executing. Therefore, this "picture" of the system is only approximate.

Example:

```
$ ps -ax
   PID TTY        TIME CMD
     0 ?       1404:51 swapper
     1 ?         8:38 /etc/init
    47 ?         8:32 /etc/update
    50 ?         1:38 /etc/cron
    89 ?         0:00 /etc/init
    96 ?         0:00 /etc/init
  8360 ?         0:00 /etc/init
  8566 ?         0:00 /etc/init
 11937 ?         0:00 /etc/init
 27349 ?         0:00 /etc/init
 26348 console   0:04 -sh
 26525 1         0:04 -sh
 28144 1         0:13 ps -ax
 26109 2         0:07 -sh
 27823 2         0:03 watch lp?
 28145 2         0:00 sleep 15
  9982 3         0:05 -sh
 25198 4         0:00 - w (getty)
 27709 5         0:00 - w (getty)
 21096 6         0:11 -sh
 28148 6         0:00 cat foo
 27507 7         0:00 - w (getty)
```

The listing shows the process ID number (PID), the number of the terminal that the process originated from (TTY), the cumulative execution time of the process (TIME), and an approximation of the process command line (CMD).

See also:

- Online information: enter **man ps**.

COMMUNICATING WITH OTHER USERS

Sending Mail to Another User

Description:

This procedure mails a message to another user.

Procedure:

$ mail **usernames**
message ... <CR>
.
.
.
message ... <CR>
^D
$

Parameters:

usernames—The login names of the users to whom you want to send mail. If you include your own login name, you will be sent a copy of the letter.

message—The letter you want to send.

Comments:

The **mail** command puts a header at the beginning of the letter showing the addressee, the sender, and the date. After you finish typing the letter, enter CTRL-D to send it.

Example:

```
$ mail horatioa
Have you received the software package from Data-mung yet?
When you do, please send me a summary.
^D
$
```

This example sends mail to horatioa.

```
$ mail gregg <letterfile
```

This example sends the contents of *letterfile* (in the current directory) to gregg.

```
$ mail ricks
Tuesday reminders: project meeting at 9am,
code walk-through at 10:30,
pick up kids at 4:30
^D
$
```

In this example, ricks sends himself mail to remind himself of the day's commitments.

Receiving and Viewing Mail

Description:

This procedure shows you how to view your mail.

Procedure:

[When mail is in your inbox, the following message appears:] You have mail.

[To read your mail, you type:]
$ mail

Example:

```
You have mail.
$ mail
From charlesd Tue Feb 17 12:35:00 1981
Have you received the software package from Data-mung yet?
When you do, please send me a summary.

? d
```

This example receives mail from another user. When the question mark prompt appears, type **d** to return to the TNIX shell.

Writing to Another User's Terminal

Description:

This procedure allows you to communicate from your terminal with another user.

Procedure:

$ write **username**
message ... <CR>
.
.
.
message ... <CR>
^D

Parameters:

username—The user to whom you want to send a message.

message—The text you want to send. Each line of text is sent to your correspondent when you type the <CR>.

Comments:

The other user must also use the **write** command to respond.

Example:

User harryw (on terminal tty2) initiates a conversation with walterp (on tty4).
Here is how the communication appears on Harry's terminal:

```
$ write walterp
What time are you going to lunch today?
Message from walterp tty4 at 11:45 ...
I'm going as soon as I finish up here--about 30 minutes.
EOF
^D
```

Here is how the communication appears to Walter:

```
$
Message from harryw tty2 at 11:45 ...
What time are you going to lunch today?
write harryw
I'm going as soon as I finish up here--about 30 minutes.
^D
EOF
```

USEFUL SYSTEM OPERATIONS

Executing a Background Program

Description:

This procedure places a command in background mode, thus allowing you to continue with other system tasks while the command is being executed.

Procedure:

$ command &

Parameters:

command—The command to be executed.

Comments:

The **&** operator places the job in background mode. The system displays the process ID number (which you can use to monitor or kill the process), then returns the prompt, indicating that you can enter other commands while the original process is executing.

Example:

```
$ pas -l bigmodule.ps >bigmodule.pl &
65
$
```

This example compiles a Pascal program in background mode. The process ID number is 65.

Aborting a Background Program

Description:

This procedure terminates a background program.

Procedure:

$ kill -9 **PID**

Parameters:

PID—The process ID number.

Comments:

The **ps** command displays the process ID number. You can specify the most recently started background job by typing "$!" rather than the process ID number.

Example:

```
$ kill -9 $!
```

This example kills the most recently started background job.

```
$ ps
   PID TTY       TIME CMD
  4061 4         0:04 ps
  4438 4         0:02 -sh
  4491 4         0:00 sleep 300
  4503 4         0:00 -sh
$ kill -9 4491
4491 Killed
$ ps
   PID TTY       TIME CMD
  4061 4         0:04 ps
  4438 4         0:03 -sh
  4522 4         0:00 -sh
```

This example terminates the process identified as PID 4491.

Redirecting Output into Another Program

Description:

This procedure pipes the standard output of one procedure into the standard input of another procedure.

Procedure:

$ **process1 ¦ process2**

Parameters:

process1—The process whose output is directed to the pipe.

process2—The process that will accept the information flowing through the pipe as standard input.

Example:

```
$ who ¦ grep cynthiat
cynthiat   tty4    Nov  6 08:10
```

This example checks to see if cynthiat is logged in, and tells you that she logged in on terminal 4 at 8:10 AM on November 6th.

The **who** command generates information about all users currently logged in, and directs it to the **grep** command as standard input. The **grep** command then looks for the string "cynthiat" and displays any lines containing that string.

Checking Disk Usage

Description:

This procedure lists the number of blocks used by each file or subdirectory in a directory.

Procedure:

$ du **directory**

Parameters:

directory—The pathname of the directory for which you want the disk usage information. If you do not specify a directory, disk usage information is provided for the current directory.

Example:

```
$ du /usr/marilynw
218     /usr/marilynw/corr
128     /usr/marilynw/c
28      /usr/marilynw/bin
546     /usr/marilynw/stress
444     /usr/marilynw/pressure/shell
996     /usr/marilynw/testcase
224     /usr/marilynw/air.raid
18      /usr/marilynw/.bin
284     /usr/marilynw/.bak
1896    /usr/marilynw
```

This example lists the number of disk blocks (by directory) used by marilynw.

Downloading a Program to an 8540

Description:

This procedure downloads an executable program from the 8560 to the 8540.

Procedure:

$ lo <**8560.file**

Parameters:

8560.file—The 8560 file to be downloaded. The file must be a load file produced by a TEKTRONIX assembler, compiler, or linker.

Comments:

The 8560 file contains information that tells the 8560 where to load the code into 8540 program memory.

Example:

```
$ lo <test.lo
```

The file *test.lo* is an 8560 file containing a program that will be downloaded into 8540 program memory. The file also contains the location in 8540 program memory where *test.lo* is to be loaded.

See also:

- Logging in Through an 8540

DISK OPERATIONS

Archiving Files to a Flexible Disk

Description:

This procedure creates an archive on a flexible disk and copies files to it.

Procedure:

[Place a formatted disk in the 8560 Flexible Disk Drive.]

$ fbr -c **name-list**

Parameters:

name-list—A list of files or directories to be archived.

Comments:

Files are recognized by string comparison. Thus, the file *lion* does not match the file */usr/marlinp/lion*, even if */usr/marlinp* is the current directory. It is recommended that you specify filenames within the current directory (rather than using full pathnames).

CAUTION

This command initializes the flexible disk before it archives the designated files and directories. Therefore, all previous information on the flexible disk is destroyed.

Example:

```
$ fbr -c 68000 research.dir 8086
```

This example initializes a flexible disk, and archives two files and one directory (with its contents). All three are in the current directory.

Adding Files to an Existing Archive on a Flexible Disk

Description:

This procedure adds files to a previously created archive.

Procedure:

$ fbr -uv **name-list**

Parameters:

name-list—A list of files or directories to be archived.

Comments:

Files are recognized by string comparison. Thus, the file *thews* does not match the file */usr/clarkk/thews*, even if */usr/clarkk* is the current directory. It is recommended that you specify filenames in the current directory (rather than using full pathnames).

A file is updated in the archive only if the file was modified since it was last archived.

Example:

```
$ fbr -uv directoryl prog2.asm
```

This example updates the archived copy of all files in the directory *directory1*, and the file *prog2.asm*.

Retrieving Files from a Flexible Disk Archive

Description:

This procedure retrieves files stored on a flexible disk archive.

Procedure:

$ fbr -vx **name-list**

Parameters:

name-list—The list of directories or files you want to retrieve from the archive and place into your directory.

Comments:

If you do not specify a file or directory, the entire contents of the archive are retrieved.

When files are retrieved from the archive, they retain the name on the archive. For instance, if an entry is named */usr/dickg/robin* on the disk, it will always be retrieved with the name */usr/dickg/robin* no matter what the current directory is.

Files are recognized by string comparison. Thus, the file *robin* does not match the file */usr/dickg/robin* even if */usr/dickg* is the current directory.

Example:

```
$ fbr -vx dir4
```

This example restores the directory *dir4*.

Deleting Files from a Flexible Disk Archive

Description:

This procedure deletes files from a flexible disk archive.

Procedure:

$ fbr -vd **archived-names**

Parameters:

archived-names—The names of the files or directories to be deleted from the archive.

Comments:

Files are recognized by string comparison. Thus, the file *sheriff* does not match the file */usr/mattd/sheriff*, even if */usr/mattd* is the current directory.

Example:

```
$ fbr -vd file1
```

This example deletes *file1* from the archive.

Listing the Files in an Archive on Your Terminal

Description:

This procedure lists the contents of a flexible disk archive on your 8560 system terminal.

Procedure:

$ fbr -tv

Example:

```
$ fbr -tv
"directory of /usr/bradl/px"
created  Jul 20 17:40:38 by bradl    ,abstract
modified Jul 20 17:48:35
   mode       uid       gid       size       modified      fblk  name

drwxrwxrwx bradl     abstract       0 Jul 20 17:39:42     7 .
-rw-r--r-- bradl     abstract     252 Nov  5 09:22:51     8 Makefile
-rw-r--r-- bradl     abstract     465 Mar  9 11:14:54     9 console.c
-rwxrwxrwx bradl     abstract    2380 Oct 17 07:52:15    11 format
-rw-r--r-- bradl     abstract    4955 Jan 28 10:43:10    28 px.c
-rwxrwxrwx bradl     abstract     402 Aug 11 09:21:48    43 pxboot
-rw-r--r-- bradl     abstract    4790 Aug 11 09:17:52    44 pxboot.s
-rwxrwxrwx bradl     abstract    2380 Oct 17 07:52:15    11 pxfmt
-rw-r--r-- bradl     abstract    3174 Oct 17 07:51:24    54 pxfmt.c
-rw-rw-rw- bradl     abstract    4480 Oct 17 07:52:50    67 pxfmt.s
   10 files used
   13 files free
   67 blocks free
```

Transferring Files to an 8550 Flexible Disk

Description:

This procedure copies 8560 files stored on the hard disk to an 8550 flexible disk inserted into an 8560.

Procedure:

$ dsc50 -w **8560source 8550dest**

Parameters:

8560source—The 8560 file(s) or directory you want to transfer to an 8550 flexible disk. If you specify a directory, all the files in that directory are copied, but not the subdirectories.

8550dest—The pathname of the 8550 file that you are writing to. If only one 8560 file is being transferred, this will be the name of a file on the 8550 flexible disk. (If this file already exists, it will be overwritten.) If more than one 8560 file or if an 8560 directory is being transferred, the destination must be a directory name.

Comments:

Because the 8560 has only one flexible disk drive, you may omit the prefix */VOL/volume-name* when specifying the 8550 file.

Example:

```
$ dsc50 -w swan lake
```

This example copies the 8560 file, *swan*, into the 8550 file, *lake* in the current directory.

```
$ dsc50 -w /usr/georgh/mess /usr/petrt/nutcrkr /comp
```

This example copies the two 8560 files */usr/georgh/mess* and */usr/petrt/nutcrkr* into the 8550 directory *comp*.

```
$ dsc50 -w /usr/johans /comp
```

This example copies the files in the 8560 directory */usr/johans* to the 8550 directory *comp*. Any subdirectories in */usr/johans* are not copied.

Transferring Files from an 8550 Flexible Disk

Description:

This procedure copies files to the 8560 hard disk from an 8550 flexible disk inserted into the 8560.

Procedure:

$ dsc50 -x **8550source 8560dest**

Parameters:

8550source—The file(s) or directory you want to copy from an 8550 flexible disk. If you specify a directory, all the files in that directory are copied, but not the subdirectories.

8560dest—The pathname of the 8560 file that you are writing to. If only one 8550 file is being transferred, this will be the name of a file on the 8560. (If this file already exists, it will be overwritten.) If more than one 8550 file or if an 8550 directory is being transferred, the destination must be a directory name.

Comments:

Because the 8560 has only one flexible disk drive, you may omit the prefix */VOL/volume-name* when specifying the 8550 file.

Example:

```
$ dsc50 -x lake swan
```

This example copies the 8550 file, *lake*, into the 8560 file, *swan*.

```
$ dsc50 -x /usr/comp/mess /usr/georgh
```

This example copies the 8550 file */usr/comp/mess* into the 8560 directory */usr/georgh*.

```
$ dsc50 -x /comp /usr/johans
```

This example copies the files in the 8550 directory *comp* to the 8560 directory */usr/johans*. Any subdirectories in *comp* are not copied.

Section 4
SHELL PROGRAMMING

ILLUSTRATIONS

TABLES

Section 4

SHELL PROGRAMMING

INTRODUCTION

This section is your reference guide to the TNIX shell—the interface between you, the TNIX operating system, and the programs that TNIX executes. This section describes the programming features of the TNIX shell, with emphasis on how you can customize your programming environment to your own special needs.

This section will not teach you how to program—to best utilize the information presented in this section, you should be familiar with a high-level programming language such as Pascal or C. Also, you should be familiar with the information presented in the *Learning Guide* and the *TNIX Operating System* sections.

In this section, we'll look at the command shortcuts that are available with the shell. We'll start with a long command line, look at how that command line can be turned into a simple shell program, then look at how that shell program can be made more versatile. Throughout this process, we'll look at ways to customize a shell program, including:

- how to manipulate shell variables,
- how to pass parameters to programs,
- how to print messages from a shell program,
- how to use the shell's structured statements: the conditional **case** and **if** statements, and the repetitive **for**, **while**, and **until** statements.

Finally, we'll look at:

- how the shell parses command lines,
- how to override the shell's treatment of special characters (such as * and !),
- some high-level concepts used by the shell, and
- a language reference summary of the shell programming language.

OVERVIEW

This overview describes the general capabilities of the TNIX shell. If you're interested in how to write a shell program, turn to the discussion on "Writing Shell Programs".

An Interactive Command Interpreter

The shell is an interactive command interpreter—it mediates communication between you and the TNIX operating system. When you type a command, then press the RETURN key, the shell analyzes the line that you typed, making any appropriate transformations (such as expanding a two-character variable name to a 60-character pathname), then executes the proper command(s). If necessary, the shell prompts you for additional input.

For an discussion of the interactive features of the shell, see the latter part of the *TNIX Operating System* section.

How to Execute the TNIX Shell

When you log in, TNIX executes the **sh** (shell) program for you. The shell then executes the commands located in the *.profile* file in your HOME directory. (The HOME directory is described in the discussion of "The Default Shell Environment Variables", later in this section.)

You can modify the way in which you communicate with TNIX by replacing the shell with a different command interpreter, such as the TEKTRONIX Keyshell interface. One way to do this is to add the command that executes the different command interpreter to your *.profile* file.

If you're using a command interpreter other than the shell but would like to use the TNIX shell, you can execute the shell from that command interpreter by typing **sh**. That's because the TNIX shell is a program.

The Shell Controls a Program's Input and Output

The shell controls the source and destination of a program's input and output. The *TNIX Operating System* section showed you how to use a file as input to a program, and how to send a program's output to a file. Later in this section, in the "Program I/O Control" discussion, we'll look at different ways to manipulate a program's input and output.

A Programming Language

The shell is also a programming language. Its program statements help to simplify routine tasks, for example, continually ringing the bell on your terminal when a large compilation completes. These program statements include:

- **if**,
- **case**,
- **while** loops,
- **until** loops,
- **for** loops,
- **read** (data input), and
- error trapping statements

The shell also supplies:

- string-valued variables,
- variable substitution,
- global and local variable designations,
- boolean operations on variables,
- prameter passing between the shell and any programs (including other shells) that the shell executes, and
- concurrent program execution.

In the following discussions, we'll look at ways to use these features of the TNIX shell when writing shell programs. Before we do that, we'll look at the types of program I/O control available with the shell.

PROGRAM I/O CONTROL

The following paragraphs summarize information about how the shell redirects a program's input and output.

Normally, the I/O redirection symbols < and > represent file descriptors 0 and 1 (standard input and output), respectively. However, by preceding the symbol with a digit, you can make it stand for any file descriptor—not just 0 and 1.
For example,

```
...2>file
```

directs the standard error output (file descriptor 2) to *file*.

>file Directs standard output to *file*, which is created if it does not already exist. Filename pattern matching and shell command line parsing does not take place, so that, for example,

```
echo Hello there! >*.message
```

creates the file **.message*.

>>file Directs standard output to *file*. If *file* exists, the output is appended to the end of *file*; otherwise *file* is created.

<file Takes standard input from *file*.

<<word Takes standard input from the lines between, but not including, the initial string specified in *word*, and a subsequent line consisting only of the string specified in *word*. If you don't want the shell to interpret any special characters, for example, **$** and **'**, enclose *word* in apostrophe (') characters.

The shell assigns the file descriptors 0, 1, and 2 to the standard input, standard output, and standard error output, respectively. The following paragraphs show how you can use these file descriptors to manipulate a program's input and output.

>&*n* Duplicates file descriptor *n* using the TNIX system call **dup**, and uses the result as the standard output. For example,

```
$ cc newton.c >logfile 2>errors
```

sends the standard output to the *logfile* file and the standard error output to the *errors* file.

```
$ cc newton.c >logfile 2>&1
```

merges the standard error output with the standard output, sending the result to the *logfile* file.

<&*n* Duplicates file descriptor *n* using the TNIX system call **dup**, and uses the result as the standard input.

<&- Closes the standard input.

>&- Closes the standard output.

WRITING SHELL PROGRAMS

Shell programs are simple to write. You'll find that a shell program with a simple mnemonic name can save you a lot of tedious typing, or can save you the time it takes to look up a certain command's syntax.

A shell program consists of any sequence of commands that you can enter from your terminal. To create a shell program,

1. store these commands in a file, then
2. make the file executable with the **chmod +x** command. For example, to make the *reminder* file executable, type:

```
$ chmod +x reminder
```

3. next, store this file in your personal programs directory.

(For information on how to set up a personal programs directory, see the *TNIX Operating System* section.)

Whenever you want to execute the commands stored in the **reminder** file, type:

```
$ reminder
```

The shell will execute the commands in the **reminder** file, as if you had typed them from your terminal.

In the following paragraphs, we'll look how you can use shell programs to simplify some of your everyday tasks.

How to Specify a Different Name For a TNIX Command

With the TNIX shell, you can use different names for the same command. Since TNIX commands names tend to be short and cryptic, you may want to use a command name that is easier to remember. For example, the command that changes the name of a file is called **mv**. What if you want a command called **rename** that performs the same function that **mv** does? There are two ways to do this:

1. You can create a *link* between the TNIX command that performs a specific function and the command name that you want to use for that function. First, determine where the specific TNIX command is located. (Most TNIX commands are located in either the */bin* or */usr/bin* directories). Next, use the **ln** command to link the TNIX command to your command name. For example, if your personal programs directory is */usr/lazaruslong/bin*, and you want TNIX to execute the **mv** command (located in */bin*) whenever you type **rename**, type the following command:

   ```
   $ ln /bin/mv /usr/lazaruslong/bin/rename
   ```

   Now, when you type, for example,

   ```
   $ rename list list.old
   ```

   the shell executes the following command:

   ```
   mv list list.old
   ```

2. If you get an error message of the sort

   ```
   /bin/mv: Cross-device link
   ```

   after typing the **ln** command, then the TNIX command that you want to link to is not on the same filesystem as your account, and the above procedure won't work. An alternative is to create an executable file that contains the text

   ```
   case $# in
   0 ) TNIX-command;;
   *) TNIX-command "$@" ;;
   esac
   ```

   where *TNIX-command* is the name of the TNIX command that you want executed when you type your custom command name. (The shell replaces **"$@"**, a shell variable, with the command line parameters to the command that you enter, and **$#** with the number of command line arguments. We'll look at how to use the **"$@"** and **$#** shell variables in the "Shell Variables" discussion, later in this section. We'll also discuss the **case** statement in the "CASE Statements" discussion, later in this section.)

   To call the **mv** command "rename", create an executable file called **rename** that contains the command lines

   ```
   case $# in
        0 ) mv ;;
        *) mv "$@" ;;
   esac
   ```

   then place this file in your personal programs directory.

How to Change the Way a Command Executes

If you don't like the default method in which a command executes, you can create your own version of that command.

First, make sure that the shell **always** checks your personal programs directory before checking the */bin* and */usr/bin* directories for a command to execute. For example, if your personal programs directory is */usr/lazaruslong/bin*, your *.profile* file (in your HOME directory) should contain the following line:

```
PATH=:$HOME/bin:$PATH; export PATH
```

(PATH is a shell environment variable, described in "The Default Shell Environment Variables" discussion, later in this section.)

Next, create an executable shell program with the same name as the TNIX command, using the format:

```
case $# in
     0) TNIX-command-pathname command-options ;;
     *) TNIX-command-pathname command-options "$@" ;;
esac
```

Here, *TNIX-command-pathname* is the pathname of the TNIX command and *command-options* are the default command options that you define. To see how this works, let's look at an example.

Suppose you want the **rm** command to ask you whether or not you want it to delete each file before it actually deletes that file (this is the **rm -i** command). To do this, create an executable file called **rm** that contains the command lines

```
case $# in
     0) /bin/rm -i ;;
     *) /bin/rm -i "$@" ;;
esac
```

Then place this file in your personal programs directory. Now, if you type

```
$ rm list.o
```

the shell executes

```
/bin/rm -i list.o
```

Making Routine Tasks Easier to Do

Earlier in this section, we saw that the shell can execute commands stored in a file, just as if you sat there and entered those commands from your terminal. This is one of the most powerful features of the TNIX shell. In the following paragraphs, we'll look at ways to create a shell program called "compile", located in your personal programs directory, that executes the command:

```
cc -o mover mover.c mover1.o mover2.o mover3.o
```

(This command uses the optional C compiler, available with the 8560 Series Native Programming Package.) The simplest approach is the one we've outlined above: create a file called *compile* that contains the above command line, then make it executable with the **chmod +x** command:

```
$ chmod +x compile
```

Now, any time that you want to execute the command

```
cc -o mover mover.c mover1.o mover2.o mover3.o
```

all you have to do is type

```
$ compile
```

Now suppose that you want to embellish the **compile** command that you've just created. For example, you may want to:

- supply it with certain command line options,
- have it check to see whether or not certain files exist before invoking the C compiler (with **cc**),
- execute certain commands if the compiler locates any errors, etc.

In the following discussions, we'll look at how to use the shell's structured statements (**for**, **while**, **until**, **if**, and **case**) and string-valued variables (and the various operations on these string-valued variables) to turn the **compile** command into a more versatile, more powerful command.

First, we'll look at how to use the shell's string-valued variables, then at how to use the shell's structured statements. We'll also see how to debug shell programs and how to create log files.

Shell Variables

You can use shell variables to:

- determine the number and values of command line arguments supplied to shell programs,
- input data into shell programs,
- generate unique filenames,
- abbreviate directory names and pathnames,
- determine whether or not a program that was executed terminated properly,
- control the default directories that TNIX expects commands to be located in, and
- store information about your programming environment, including the type of terminal that you are using, and the target microprocessor that you are working with.

Valid Names for Shell Variables

Variable names begin with a letter or an underscore character, and consist of letters, digits, and underscores. Table 4-1 shows examples of valid and invalid variable names.

Table 4-1
Valid and Invalid Variable Names

Valid	Invalid	Invalid Because
Temp_Name	Temp-Name	illegal character (-)
_another	another.one	illegal character (.)
day10	10day	begins with a number

Substituting a Variable's Value For Its Name

To substitute a variable's value for its name, precede the name with a **$**. Let's look at the **echo** command to see how this works.

The **echo** command prints its command line arguments, for example,

```
$ echo Hello there!
Hello there!
```

If you assign the words "Hello there!" to a variable called "message" by typing

```
$ message="Hello there!"
```

you can use the **echo** command to print the value of the "message" variable:

```
$ echo $message
  Hello there!
```

Or, you can assign the pathname */usr/lazaruslong/first.book/chapter1* to the variable called "c1" by typing:

```
$ c1=/usr/lazaruslong/first.book/chapter1
```

Now, typing

```
$ cd $c1
```

is equivalent to typing

```
$ cd /usr/lazaruslong/first.book/chapter1
```

An alternate notation is used when a variable name is followed by a letter, number, or underscore. In this case, you enclose the variable name within "{ }" characters, to differentiate the variable name from the character that follows. The following example shows how the "{ }" characters are used:

```
$ tmpa=/tmp/tempfile
$ tmp=/tmp/temp
$ echo $tmpa
/tmp/tempfile
$ echo ${tmp}a
/tmp/tempa
```

Note that the value of the variable **${tmp}a** is "/tmp/tempa", whereas the value of the variable **tmpa** is "/tmp/tempfile".

Assigning Values to Variables

There are two ways to assign values to variables:

1. You can type the variable name followed by an equals sign (=) and the value of the variable. For example:

```
$ TempFile=a.tmp ListFile=a.list LogFile=a.log
```

NOTE

Do not put any spaces between the variable name, the equals sign, and the variable's value.

2. You can use the **read** statement to input values to variables, for example:

```
$ read TempFile ListFile LogFile
a.tmp a.list a.log          [ this is on a new line ]
```

This example performs the variable assignments as in step 1. If you specify more words in the input line than variables in the **read** statement, the extra words are assigned to the last variable in the **read** statement. To see how this works, let's look at an example:

```
$ read a b
this is a line of text
$ echo $a
this
$ echo $b
is a line of text
```

You can assign a null value to a variable by typing a space, tab, or newline character after the equals sign (=) character, for example:

```
$ flags= files= printers=
```

(You enter a newline character by pressing the RETURN key.) In this example, the variables "flags", "files", and "printers" are assigned null values.

If you want to assign text that contains one or more space, tab, or newline characters, or any of the shell's special characters to a variable, you should enclose the text to be assigned to the variable within apostrophe (') characters. For example, to assign the two lines

```
This line contains the *, ", ?, and newline characters
and this second line contains the ! and | characters
```

to the variable "message", type:

```
$ message='This line contains the *, ", ?, and newline characters
> and this second line contains the ! and | characters'
```

If you want to assign an apostrophe to a variable, enclose it in quotation (") marks. For example, to assign the line

```
That won't do!
```

to the variable "message", type:

```
$ message="That won't do!"
```

A note about using quotation (") marks and apostrophe (') characters: the shell does not substitute a value for a variable name, or a command's output for a command enclosed in accent grave (`) characters, if the variable or accent grave-enclosed command is enclosed in apostrophe (') characters. However, variable and/or command substitution will occur if a variable or accent grave-enclosed command is enclosed in quotation (") marks.

Concatenating Shell Variables

You can concatenate strings or variables onto an existing variable. For example, suppose the value of the PATH variable is ":/bin:/usr/bin", and you want to concatenate the string ":/usr/lazaruslong/bin" onto the beginning of the PATH variable. To do this, type:

```
$ PATH=:/usr/lazaruslong/bin$PATH
```

Or, you can concatenate the string ":/usr/lazaruslong/bin" onto the end of the PATH variable by typing:

```
$ PATH=$PATH:/usr/lazaruslong/bin
```

Similarly, to concatenate the value of the "c" variable onto the beginning of the "a" variable, type:

```
$ a=$c$a
```

Or, you can concatenate the value of the "c" variable onto the end of the "a" variable by typing:

```
$ a=$a$c
```

Assigning a Command's Output to a Variable

You can assign a command's output to a shell variable by enclosing the command within accent grave (') characters. For example, suppose you are working in a directory with a long pathname. After working in another directory, you want to be able to return to this directory, without typing the entire pathname of the directory. The following example shows how to assign the name of the current directory to the "cur" variable:

```
$ pwd
/usr/lazaruslong/newprojects/magicwand/source
$ cur='pwd'
$ echo $cur
/usr/lazaruslong/newprojects/magicwand/source
```

Now, whenever you type

```
$ cd $cur
```

the following command is executed:

```
$ cd /usr/lazaruslong/newprojects/magicwand/source
```

Assigning Default Values to Variables

You can assign a default value to a shell variable. Normally, if a shell variable is not set, the null string is its default value. However, there are other ways to assign default values to variables, depending on whether or not a variable has been previously assigned a value (including the null value).

One common use for default variable assignments is in shell programs, where the presence or absence of certain command line arguments determines what commands will be executed by the shell program. For example, earlier in this section, we described a **compile** command that executes the command

```
cc -o mover mover.c mover1.o mover2.o mover3.o
```

whenever you type

```
$ compile
```

Because you probably compile other programs than just **mover**, you can make your **compile** command more useful by changing the text of the **compile** command so that the word "mover" is replaced by the first command line argument that you supply. (The "Interpreting Command Line Arguments in Shell Programs" discussion, later in this section, shows how to do this.) In the following paragraphs, we'll look at the different ways to specify default values for shell variables. We'll use the following notation:

- *X* is the variable to be assigned a value
- *varname* is the name of the variable being tested for a default value
- *value* can be text, another variable, the output of a list of commands enclosed in accent grave (') characters, or any combination of these.

X=${*varname*-*value*} Assigns *value* to *X* if *varname* has not been assigned any value; otherwise, assign the value of *varname* to *X*.

X=${*varname*=*value*} Assigns *value* to *varname* if *varname* has not been assigned any value, then assigns the value of *varname* to *X*.

X=${*varname*+*value*} AsAssigns *value* to *X* if *varname* has been assigned a value; otherwise, does not assign a value to *X*.

X=${*varname*?*value*} Assigns *value* to *X* if *varname* has been assigned a value. Otherwise, prints *value* or the message "*varname*: parameter not set" if *value* is not specified.

Examples. The command

```
$ a=${d-$c}
```

assigns the value of "d" to variable "a" if "d" is set; otherwise, it assigns the value of "c" (if any) to variable "a". For example,

```
$ DefaultFlag=-x
$ flag=${InputFlag-$DefaultFlag}
```

assigns the value of the "DefaultFlag" variable (-x) to the variable "flag" only if the "InputFlag" variable has not been set. Note that the null value will be assigned to the "flag" variable if the "InputFlag" variable has been set to the null value.

Alternatively, the default value for the "flag" variable can be set with

```
$ flag=${InputFlag--x}
```

which assigns the value **-x** to the "flag" variable if the "InputFlag" variable is not set.

If you want to print the error message "InputFlag: variable isn't set!" instead of assigning a default value to the "InputFlag" variable, you can type:

```
$ flag=${InputFlag?"variable isn't set!"}
```

If you don't supply a message, as in

```
$ flag=${InputFlag?}
```

the default message is

```
InputFlag: parameter not set
```

Global and Local Variables

You can use the **export** command to make the value of a shell variable available to any program that the shell executes. If you don't use the **export** command, a variable's value remains *local* to the shell and is not accessed by any programs that the shell executes.

For example, if you are using the TEKTRONIX Language-Directed Editor (LDE), you use the **export** command to set and specify as global a TERM variable before executing LDE. The following command sets the TERM variable to "4105", then makes it a global variable with the **export** command:

```
$ TERM=4105; export TERM
```

Global variables can be reset by programs executed by the shell, but will revert to their original values when the executing program terminates.

The Default Shell Environment Variables

Certain shell variables are set each time you log into TNIX. These variables are called environment variables, because they control how the shell functions.

Here is a list of the standard shell environment variables that are set each time you log in:

HOME The shell assigns your HOME or *login* directory to this variable. When you type a **cd** command with no parameters, the shell executes a **cd $HOME** command. Your login directory is specified in the sixth field in the */etc/passwd* file. (Fields are separated by a colon.) For example, to find out what lazaruslong's login directory is, type:

```
$ fgrep lazaruslong /etc/passwd
lazaruslong:MMwYQYpdkxTMQ:276:175::/usr/lazaruslong:
```

this is lazaruslong's HOME directory

In general, if your login directory is */usr/lazaruslong*, the default value of the variable HOME is:

```
HOME=/usr/lazaruslong
```

IFS — The internal field separator. The shell separates arguments to programs (that is, it parses the command line) based on the characters assigned to this variable. Normally, the ASCII space, tab, and newline (octal 12) characters are assigned to this variable.

PATH — When you ask the shell to execute a program, the shell looks for that program in the directories specified by the value of the PATH variable. If the value of the PATH variable starts with a colon, then the first field specifies a null value, which corresponds to the current directory. (The colon (:) character separates each directory name.) The default value is:

```
PATH=:/bin:/usr/bin
```

In this case, the shell looks for commands to execute in your current directory (because the PATH variable begins with the colon character), then in the */bin* directory, then in the */usr/bin* directory. The shell prints an error message if it cannot find the command that you want to execute.

PS1 — The shell's primary prompt string, usually set to "$ ". The following line changes your prompt to "Type something! ":

```
$ PS1="Type something! "; export PS1
```

PS2 — The shell's secondary prompt string, usually set to "> ". The shell displays this prompt when it needs more information before it can execute a command, for example, when a command extends over two or more lines. The following line changes your secondary prompt to "Keep typing! ":

```
$ PS2="Keep typing! "; export PS2
```

Standard, Automatically Updated Shell Variables

Certain shell variables are automatically updated each time the shell executes a command. These variables are described in the following paragraphs.

$?** — The shell assigns the exit status of the last command executed to the **$? variable. This command may have been running in the background. Most commands return a zero exit status if they terminate successfully; otherwise, they return a non-zero exit status. The exit status returned by a command is used by the **if**, **until**, and **while** statements to determine which parts of the statement to execute. (These statements are described in the "Structured Statements" discussion, later in this section.) For further information, see the "Setting the Exit Status of a Program" discussion, later in this section.

$# — The number of command line arguments or positional parameters, in decimal.

$$ — The process number, in decimal, of the currently executing shell. Since each existing process has a unique process number, this string can be used to generate unique temporary filenames. For example, the statement

```
TempFile=/tmp/ex.$$
```

generates the unique filename */tmp/ex.***N**, where **N** is some decimal number.

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

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

Interpreting Command Line Arguments in Shell Programs

One way to make a shell program more versatile is to allow it to access its command line arguments. A shell program can access its command line arguments, or positional parameters, as the values of the variables shown in the following list:

$# The number of arguments to the program

$0 The name of the program

$1, **$2**, ..., $*n* The first argument to the program, the second argument to the program, and so on

"$*"** All arguments to the program are interpreted as one string. Thus, **"$*" (the quotation marks are required) is equivalent to:

```
"$1 $2 $3 $4 ..."
```

"$@"** All command line arguments are passed to another command without checking for argument separators. Thus, **"$@" (the quotation marks are required) is equivalent to:

```
"$1" '$2" "$3" "$4" ...
```

For example, assume that a program called **delete** contains the line

```
rm "$*"
```

When you type

```
$ delete m1.o m2.o m3.o
```

the shell executes the command

```
rm "m1.o m2.o m3.o"
```

attempting to delete a single file called "*m1.o m2.o m3.o*".

To pass command line arguments to a program, use the **"$@"** notation instead of the **"$*"** notation. The following examples show why.

Let's assume that a program called **delete** contains the line

```
rm "$@"
```

When you type

```
$ delete m1.o m2.o
```

the shell executes the command

```
rm m1.o m2.o
```

On the other hand, when you type

```
$ delete "funny file" m1.o m2.o
```

the shell executes the command

```
rm "funny file" m1.o m2.o
```

attempting to delete the files *m1.o*, *m2.o*, and *funny file*, whose filename contains a space character. (You can't do this with the "**$***" notation.)

You may want a shell program to transform its command line arguments in some manner. (Figure 4-4, later in this section, shows an instance where a shell program transforms its command line arguments.) There are two ways to change the values and number of command line parameters within a shell program:

- The **shift** command assigns the values of **$2** to **$1**, **$3** to **$2**, and so on, and decrements the **$#** variable (the number of command line arguments) by one.
- The **set** command resets both the number (the value of **$#**) and values (**$1**, **$2**, ..., $*n*) of the command line arguments. The following example shows how to do this:

```
$ set one two three
$ echo $1 $2 $3 $#
one two three 3
```

An Example. Earlier in this section, we described the **compile** command, a shell program containing the following text:

```
cc -o mover mover.c mover1.o mover2.o mover3.o
```

We also saw that you can specify default values for command line arguments to a shell program. (The command line arguments are accessed as the values of the **$1**, **$2**, ..., $*n* variables, where **$1** is the first command line argument, **$2** the second command line argument, and so on.) Now let's use these techniques to make the **compile** command more versatile.

For example, you can substitute **${1}** (the value of the first command line argument to **compile**) for the word "mover" in the text of **compile**:

```
cc -o ${1} ${1}.c ${1}1.o ${1}2.o ${1}3.o
```

Then, whenever the **compile** command is executed, **${1}** is replaced by the first command line argument to **compile**. Thus,

```
$ compile lister
```

executes the command

```
cc -o lister lister.c lister1.o lister2.o lister3.o
```

However, you've lost the ability to execute the command

```
cc -o mover mover.c mover1.o mover2.o mover3.o
```

by simply typing

```
$ compile
```

To remedy this problem, you can substitute **${1-mover}** for the word "mover". Now, if no command line argument is supplied to **compile**, **${1-mover}** is replaced by the word "mover":

```
cc -o ${1-mover} ${1-mover}.c ${1-mover}1.o ${1-mover}2.o ${1-mover}3.o
```

Now, when you type

```
$ compile
```

the command

```
cc -o mover mover.c mover1.o mover2.o mover3.o
```

is executed. However, if you type

```
$ compile lister
```

the command

```
cc -o lister lister.c lister1.o lister2.o lister3.o
```

is executed. In the following discussion of conditional and repetitive program statements, you'll see ways to make the **compile** program even more versatile.

Structured Statements

The shell allows you to use the following conditional, repetitive, and data input program statements:

- conditional statements: **case** and **if** statements
- repetitive statements: **for**, **while**, and **until** statements
- data input statements: the **read** statement

A conditional statement executes no more than one of its component statements, based on the results of a conditional test. The shell's conditional constructs are IF and CASE.

A repetitive statement specifies that a list of commands in the body of the statement may be executed repeatedly. The shell's repetitive constructs are FOR, WHILE, and UNTIL.

The following pages describe each of these constructs, and shows ways to use them in shell programs.

IF Statements

An **if** statement may be in one of the following formats:

```
(1)   if   command-list1
      then command-list2
      fi


(2)   if   command-list1
      then command-list2
      else command-list3
      fi
```

```
(3)  if   command-list1
     then if   command-list2
          then if   command-list3
               then command-list4
               else command-list5
               fi
          else command-list6
          fi
     else command-list7
     fi

(4)  if   command-list1
     then command-list2
     else if   command-list3
          then command-list4
          else if   command-list5
               then command-list6
               else command-list7
               fi
          fi
     fi

(5)  if   command-list1
     then command-list2
     elif command-list3
          then command-list4
          elif command-list5
               then command-list6
               else command-list7
     fi
```

In each of the above formats, *command-list* represents a list of one or more commands. If the *last* command in the *command-list* following the **if** statement returns a true value (zero exit status), the **$?** shell variable is set to zero and the *command-list* following the **then** statement is executed. Otherwise, the *command-list* following the **else** statement (if there is an **else** statement) is executed. The reserved word **fi** terminates the **if** statement.

As formats 3 and 4 demonstrate, the *command-list* can include additional **if** statements of the type shown in formats 1 and 2.

Format 5 is equivalent to format 4.

A final note regarding the **if** construction. The sequence

```
if   command-list1
then command-list2
fi
```

may be written

```
command-list1 && (command-list2)
```

Conversely, the sequence

```
if    command-list1
then   true
else  command-list2
fi
```

may be written

```
command-list1 || (command-list2)
```

In each case, the exit status of the last simple command executed in *command-list1* determines whether or not *command-list2* is executed.

The TRUE and FALSE Commands. The **true** command returns a true value (zero exit status); the **false** command returns a false value (non-zero exit status). Thus, the statement

```
if   true
then echo Hello!
else echo Goodbye!
fi
```

always executes the **echo Hello!** command, whereas the statement

```
if   false
then echo Hello!
else echo Goodbye!
fi
```

always executes the **echo Goodbye!** command.

The TEST Command—Evaluating Boolean Expressions. Many programming languages allow statements of the form

```
if    a_certain_boolean_condition_is_satisfied
then  execute_some_commands
else  execute_some_other_commands
```

Although the TNIX shell does not perform boolean evaluations, you can use the TNIX **test** command in conjunction with the shell's structured statements to perform the same sort of boolean evaluation.

The **test** command evaluates a boolean expression, then returns a true value (zero exit status) if the value of the expression is true, or a false value (non-zero exit status) if the value of expression is false. The **test** command returns a false value (non-zero exit status) if you do not specify an expression for **test** to evaluate.

The following arguments are used to construct the expression for **test**.

-r *file*	true if the *file* exists and is readable.
-w *file*	true if the *file* exists and is writeable.
-f *file*	true if the *file* exists and is not a directory.
-d *file*	true if the *file* exists and is a directory.

-s *file*	true if the *file* exists and has a size greater than zero.
-t *number*	true if the open file with file descriptor *number* (1 by default) is associated with a terminal device.
-z *string*	true if the length of *string* is zero.
-n *string*	true if the length of *string* is non-zero.
string1 = *string2*	true if the values of *string1* and *string2* are equal.
string1 **!=** *string2*	true if the values of *string1* and *string2* are not equal.
string	true if *string* is not the null string.
n1 **-eq** *n2*	true if the integers *n1* and *n2* are algebraically equal. Any of the comparisons **-ne** (not equal), **-gt** (greater than), **-ge** (greater than or equal to), **-lt** (less than), or **-le** (less than or equal to) may be used in place of **-eq**.

These arguments may be combined with the following operators:

!	unary not operator (unary negation)
-a	binary *and* operator
-o	binary *or* operator
(*expression* **)**	used to group statements together.

You can use the **if** statement in conjunction with the **test** command to test for the existence of a file. In the following shell program, the **test** statement controls the output:

```
if    test -r logfile -a -f logfile
then  cat logfile
elif  test -f logfile
      then  echo "logfile is not readable!"
      elif  test -d logfile
            then  echo "logfile is a directory!"
            else  echo "logfile does not exist!"
fi
```

This example

- prints the *logfile* file, if that file exists and is readable; or
- prints a message saying that *logfile* exists but is not readable; or
- prints a message saying that *logfile* is a directory; or
- prints a message saying that *logfile* doesn't exist.

CASE Statements

A **case** statement may be in either of the following formats:

```
(1)   case index-value in
          case-label-list1 ) command-list1;;
          case-label-list2 ) command-list2;;
          ...
      esac


(2)   case index-value in
          case-label-list1 ) command-list1;;
          case-label-list2 ) command-list2;;
          ...
          *) command-listN;;
      esac
```

The **case** statement evaluates the value of the *index-value* (or case index), then executes the *command-list* whose case label matches the value of the case index. A case label can consist of any list of characters; however, the shell's pattern-matching characters will be interpreted by the shell as part of a regular expression.

The reserved word **esac** terminates the **case** statement.

The *case-label-list* consists of one or more case labels, separated by ! characters, and separated from the *command-list* by a close parenthesis **)** character.

Since the * shell metacharacter matches any list of characters, the

```
*) command-list;;
```

part of a **case** statement functions as an "otherwise" expression. The *command-list* associated with the * case label is executed if the value of the case index does not appear in any of the preceding *case-label-lists*. If it is used, this statement must be the last case label in the case statement.

To override the shell's processing of special pattern-matching characters such as * and **[**, precede these characters with a "\" character. (See the "Quoting—Overriding the Interpretation of Special Characters" discussion, later in this section, for further information.) In the following example, the first case label matches a case index (the value of the "i" variable) equal to **?**, the second case label matches any single-character case index, and the last case label matches any case index *not* matched by the preceding case labels:

```
case $i in
     \? ) command-list1;;
      ? ) command-list2;;
      *) command-list3;;
esac
```

You can use the **case** statement to check that you've typed a certain number of command line arguments and print an error message if you haven't. For example, you can write the **compile** command, developed earlier in this section, in the following form:

```
case $# in
     0|1 ) cc -o ${1-mover} ${1-mover}.c ${1-mover}1.o ${1-mover}2.o ${1-mover}3.o;;
      *) echo "syntax: $0 [program-name]" ;;
esac
```

When you type

```
$ compile lister
```

the **$#** variable is set to "1" (the shell sets the **$#** variable to the number of command line arguments), and the program executes the command:

```
cc -o lister lister.c listerl.o lister2.o lister3.o
```

On the other hand, if you type

```
$ compile
```

the **$#** variable is set to "0", and the program executes the command:

```
cc -o mover mover.c moverl.o mover2.o mover3.o
```

If the number of command line arguments supplied to **compile** is other than zero or one, the following message is printed:

```
syntax: compile [program-name]
```

WHILE Statements

A **while** statement has the following format:

```
while command-listl
do
      command-list2
done
```

The **while** statement evaluates the exit status of the last command executed in *command-list1*, then executes *command-list2* if the last command in *command-list1* returns a true value (zero exit status). This process is repeated as long as the last command in *command-list1* returns a true value (zero exit status).

The reserved word **do** separates *command-list1* from the body of the **while** statement, *command-list2*.

The reserved word **done** terminates the **while** statement.

Example. In the following **while** statement, the number of times that *command-list* is executed is determined by the number of command line arguments. If there are five command line arguments, then *command-list* is executed five times.

```
while test $# -gt 0      : are there more than 0 arguments?
do
      command-list
      shift              : decrement $# by 1 and
                         : rename $2 to $1, $3 to $2, etc.
done
```

(The **test $# -gt 0** command returns a true value, or zero exit status, so long as the arithmetic value of **$#** is greater than 0, that is, so long as there is at least one positional parameter.)

UNTIL Statements

An **until** statement has the following format:

```
until command-list1
do
      command-list2
done
```

The **until** statement evaluates the exit status of the last command executed in *command-list1*, and repeats the execution of *command-list2* as long as the last command in *command-list1* returns a false value (non-zero exit status).

The reserved word **do** separates *command-list1* from the body of the **until** statement, *command-list2*.

The reserved word **done** terminates the **while** statement.

Note that the **until** statement differs significantly from the way in which a **repeat/until** statement is executed in Pascal. In Pascal, the body of the **repeat/until** statement is executed first, so that it is always executed at least once. In the shell, however, the body of the **until** statement (*command-list2*) is executed only if the last command in *command-list1* returns a false value (non-zero exit status), so it's possible that the body of the shell's **until** statement may not be executed.

One use for the **until** statement is to wait until some external event occurs, then—when that event occurs—execute one or more commands. For example,

```
until test -f logfile
do
      sleep 300
done
echo 'file: logfile has arrived!"
```

executes a **sleep 300** command (pause for 300 seconds) until *logfile* exists, at which time it will print the message:

```
file: logfile has arrived!
```

Each time through the **until** statement, a five-minute pause (or **sleep**) precedes the next repetition of **test**. (Presumably, another process will eventually create the file.)

Forever Loops. A forever loop is a program statement that executes the body of the statement continuously, until some external event (such as typing a CTRL-C) aborts the program statement. You can use the **true** command with the **while** statement to execute a forever loop. For example, the following command continuously executes the **ps** command followed by a **sleep 60** command:

```
while true
do
      ps
      sleep 60
done
```

You could perform the same function by using the **false** command with the **until** statement:

```
until false
do
      ps
      sleep 60
done
```

FOR Statements

A **for** statement executes a statement repeatedly while a progression of values is assigned to a control variable in the statement.

A **for** statement may be in one of the following formats:

```
(1)   for control-variable in val1 val2 ...
      do
          command-list
      done


(2)   for control-variable
      do
          command-list
      done
```

The reserved word **in** precedes the list of values that are assigned in sequence to *control-variable*. The reserved word **do** precedes *command-list* (the body of the **for** statement). The reserved word **done** terminates the **for** statement.

In the first format, the **for** statement assigns *val1* to *control-variable*, executes *command-list*, then assigns *val2* to *control-variable* and again executes *command-list*, repeating this process until the entire list of values following the reserved word **in** has been exhausted.

The second format is equivalent to:

```
for control-variable in "$@"
do
    command-list
done
```

In this form of the **for** statement, *command-list* is executed once for each command line argument, with each command line argument assigned in turn to *control-variable*.

The following rules apply to **for** statements:

- If the reserved word **in** is included in the **for** statements, the reserved word **do** must be preceded by a newline or semicolon.
- The reserved word **done** must be preceded by a newline or semicolon.
- The *control-variable* is set to the words *val1*, *val2*, ..., in turn, with the *command-list* executing once for each value. The current value of this variable is available within the *command-list* as **$***control-variable*.
- If "**in** *val1 val2* ..." is omitted, then the loop executes once for each command line argument (positional parameter); that is, the shell assumes:

```
for control-variable in "$@"
```

- The *command-list* is a sequence of one or more simple commands separated and terminated by a newline or semicolon.

Let's look at an example of a shell program that uses a **for** loop. The program **create** contains the following text:

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

If you type:

```
$ create alpha beta
```

two files, *alpha* and *beta*, are created as empty files. (The notation, ">file", may be used by itself to create or clear the contents of a file.)

The following example prints the first ten lines of each file in a directory, asking you to press the RETURN key between files:

```
for i in *
do
    head -10 $i
    echo -n "Press RETURN to continue: "
    read x
done
```

The following examples perform the above function in a slightly different manner, illustrating different aspects of the shell:

```
(1)  for i in `ls`
         head -10 $i
         echo -n "Press RETURN to continue: "
         read x
     done


(2)  set *
     for i
     do
         head -10 $i
         echo -n "Press RETURN to continue: "
         read x
     done
```

The **for** statement is also used for iterative processes, such as repeating a certain command ten times. The following example prints the message "time to go home" on your terminal once a minute, for ten minutes:

```
for i in 1 2 3 4 5 6 7 8 9 10
do
    echo time to go home
    sleep 60                    : pause for 60 seconds
done
```

Here's another example of **for** statements. The shell program **arguments** contains the text:

```
flags= files=
for i
do
    case $i in
        -*) flags="$flags $i";;
        *)  files="$files $i";;
    esac
done
echo "$0: $flags $files"
```

This program shows one of the methods commonly used to build a command line within a shell program. Here's how this program works:

The **for** statement executes the **case** statement once for each command line argument. (Each command line argument is assigned in sequence, for each iteration of the **for** loop, to the "i" variable, the value of which then becomes the case index for the case statement.) If the first character of a command line argument is the – character, that argument is concatenated onto the end of the "flag" variable, preceded by a space. Otherwise, the command line argument is concatenated onto the end of the "files" variable, preceded by a space. For example, if you type

```
$ arguments -a -b listfile logfile
```

the **$0** variable is set to the name of the program (**arguments**), the "flag" variable is set to " -a -b", the "files" variable is set to " listfile logfile", and the **arguments** program prints

```
arguments:  -a -b listfile logfile
```

Data Input—The READ Statement

The **read** statement reads one line from the standard input. The following example shows how the **read** statement is used:

```
$ read x
this is the line being read into the "x" variable
$ echo $x
this is the line being read into the "x" variable
```

The **read** statement can also be used to assign several variables at once:

```
$ read x y z
this is the line being read
$ echo $x
this
$ echo $y
is
$ echo $z
the line being read
```

A **while** or **until** statement can be combined with a **case** statement to verify that a proper response has been typed. In the following examples, the "Thank you" message is printed only when you type a word that starts with the letters "y", "Y", "n", or "N". The following example shows how to use a **while** statement to accomplish this:

```
while
     echo -n "Please type yes or no: "
     read x
     case $x in
          [yYnN]*) false;;
          *) true;;
     esac
do
     echo You did not type a proper response
done
echo Thank you
```

The following example shows how to use an **until** statement to accomplish the same thing:

```
until
     echo -n "Please type yes or no: "
     read x
     case $x in
          [yYnN]*) true;;
          *) false;;
     esac
do
     echo You did not type a proper response
done
echo Thank you
```

The following example shows how to assign individual lines from a file to the variable "x", one line at a time. Each line in *file* is available for processing by *command-list* as the value of the variable "x". (This loop terminates after the last line of *file* is read into the "x" variable.)

```
cat file |
    while read x
    do
          command-list
    done
```

Data Output—The ECHO Command

The **echo** command prints its arguments to the standard input. Throughout this section, we've looked at how to use the **echo** command to display a variable's value, and to print messages from a shell program. In the preceding discussion, "Data Input—The READ Statement", we saw how to use the **echo** statement, along with the **read**, **until**, **case** and **while** statements, to ask a user to enter some data.

The **echo** command can also be used to print the shell's special characters:

```
$ echo 'Here is an asterisk (*), question mark (?), newline
> and bracket ([) character'
Here is an asterisk (*), question mark (?), newline
and bracket ([) character
```

For additional information on using the **echo** command to print the shell's special characters, see the "Quoting—Overriding the Interpretation of Special Characters" discussion, later in this section.

Error Handling

In the previous paragraphs, we looked at how to handle erroneous input. Sometimes, you want to make sure that a certain program is not halted while executing, or that certain temporary files are removed when a program finishes executing. The **trap** statement is used in these situations.

The **trap** statement may be in one of the following formats:

```
(1)   trap "command-list" signal1 signal2 signal3 ...

(2)   trap "" signal1 signal2 signal3 ...

(3)   trap signal1 signal2 signal3 ...
```

If the first argument to **trap** is a list of one or more commands, as in format 1, then these commands are executed whenever one of the specified *signals* is received by the shell. (A signal is an integer in the range 0—15; these signals are listed in Table 4-2.)

If *command-list* is the null string, as in format 2, the list of signals specified in this form of the statement is ignored by the shell. (This is one way to make sure that a shell program doesn't halt when you type a CTRL-C or CTRL-D.)

If *command-list* is not specified, as in format 3, then any signals that were previously specified in a **trap** statement are reset to their initial values (when the current shell process was originally invoked).

For example, to make sure that a program continues executing even if you turn your terminal off or type a CTRL-C or CTRL-D, place a **trap** statement with a null first argument, and signals 1 and 2 for the second and third arguments, at the beginning of the program. (Signal 1 corresponds to the *hangup* signal, generated when you type a CTRL-D or when you turn your terminal off; signal 2 corresponds to the interrupt signal, generated when you type a CTRL-C.) Here's an example:

```
trap "" 1 2
for i in 1 2 3
do
    echo $i
    sleep 5
done
```

In this example, the **for** statement sets the "i" variable to 1, then to 2, then 3. Each time the **for** statement assigns a value to "i", it executes the **echo** command, which prints the value of "i", then pauses for 5 seconds. The **trap** statement ensures that you won't halt the **for** loop by typing a CTRL-C or CTRL-D, or by turning your terminal off.

If a shell program creates temporary files, the program should delete these files before it finishes executing. The following example shows how to do this:

```
TempFile=/tmp/EX$$
trap "rm -f $TempFile*; exit" 0 1 2 13 15
rm -f $TempFile
for i
do
    grep $i listfile >> $TempFile
done
sort $TempFile
```

Table 4-2
Some TNIX Signals

Signal Number	Name
0	Normal program termination
1	Hangup
2	Interrupt
3	Quit
9	Unstoppable kill
13	Error when writing to a pipe
14	Alarm clock
15	Software termination [a]

[a] **Produced by the kill command**

How the Shell Parses Command Line Arguments

The shell uses the characters assigned to the IFS variable to parse command line arguments. Normally, the ASCII space, tab, and newline (octal 12) characters are interpreted as command argument separators. The shell also uses the newline and semicolon characters to separate command lines. To pass a single argument that contains one or more of these characters to a command, you can:

- enclose the argument in apostrophe (') characters; or
- enclose the argument in quotation (") marks; or
- "escape" the special character by preceding it with a backslash (\) character.

If you inadvertently create a file whose filename contains a special character, such as a space or asterisk character, and you want to remove that file, you can enclose the filename in apostrophe characters. For example, to delete a file whose name is "a *file", type:

```
$ rm -i 'a *file'
```

(Use the **-i** command option so that **rm** will ask you if you really want to delete the file.)

You can supply a null argument to a command by typing two quotation marks, one directly after the other, for the null argument. For example, to specify null arguments as the first two arguments to the **asm** command, type:

```
$ asm "" "" light.asm
```

In the following discussion, we'll describe when to use quotation (") marks, and apostrophe (') and backslash (\) characters.

Quoting—Overriding the Interpretation of Special Characters

Certain characters, such as

```
* ? : & ; $ ' ' " ( ) ( ) [ ] > < \ /
```

have special meanings when interpreted by the shell. There are three ways to override the shell's interpretation of special characters:

\char A single character *char* is interpreted literally by the shell when it is preceded by a "\" character. Here's an example:

```
$ echo \*
*
```

'text ' *Text* is passed without any interpretation, for example:

```
echo '$***'
$***
```

"Text" *Text* is passed as one argument to a command, but the shell interprets the special characters **$** and **‘**. To override the shell's interpretation of special characters enclosed in quotation (") marks, precede each special character with a "\" character. For example, the command

```
$ echo "$PATH `ls`"
```

passes the value of the PATH variable and the output of the **ls** command to the **echo** command as one command line argument. However, the command

```
$echo "*"
*
```

passes the * character, unaltered, as a command line argument to **echo**, as shown.

EXAMPLES

The following paragraphs describe some shell programming examples that use the techniques discussed in this section. If you don't understand a specific example, read over the related discussion in this section.

An Example—The Search Command

Suppose you want to find every file in a specific directory hierarchy that contains a specific sequence of characters. The **search** program shown in Figure 4-1 performs this task.

```
case $# in
    1) find `pwd` -type f -name \"*$1"\* -print ;;
    2) cd $1
       find `pwd` -type f -name \"*$2"\* -print ;;
    *) echo "syntax: $0 [initial directory] filename" ;;
esac
```

Fig. 4-1. The search command.

When you type

```
$ search asm
```

the **$#** variable is set to "1", and the following command is executed:

```
find `pwd` -type f -name \"*asm"\* -print
```

When you type

```
$ search /usr asm
```

the **$#** variable is set to "2", and the following command is executed:

```
cd /usr
find /usr -type f -name \"*asm"\* -print
```

If you type the **search** command with no arguments or with three or more arguments, the following message is displayed:

```
syntax: search [initial directory] filename
```

An Example—The Touch Command

The **touch** command, which updates the "last modified" time for a list of files, illustrates the use of **if**, **case**, and **for** statements. Figure 4-2 shows the text of the **touch** command.

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

Fig. 4-2. The touch command.

Here is a line-by-line explanation of the code:

flag=	The "flag" variable is initially set equal to the null string. It will be used to indicate whether the **–a** flag is set. The **–a** flag in this program forces files to be created if they do not already exist. Otherwise, if the file does not exist, an error message appears.
for i...do	The loop executes once for each filename or flag given as an argument to **touch**.
-a) flag=-a	If the **–a** argument is given with **touch**, then "flag" is set equal to the non-null string **–a**.
*	**touch** treats all arguments other than **–a** as filenames.
if test -f $i	Returns *true* (zero exit status) if the file named by the value of "i" exists and is not a directory.
then ln $i...	The **ln** and **rm** commands establish a link to the file and then remove it, thereby changing the "last modified" date for the file.
elif test...	Returns *true* (zero exit status) if "flag" is not the null string; so, if the file did not exist, and if "flag" does not equal the null string (that is, if the **–a** flag *did* accompany **touch**)...
then >**$1**	...create the file by assigning it to the standard output.
else echo...	If the file did not exist, and "flag" equals the null string (because **–a** was not given), then print a message.

An Example—A Modified Remove Command

This example shows a program that interactively deletes files and allows you to recover a file after deleting it.

Figure 4-3 shows the text of the **delete** command.

```
if   test ! -d "$HOME/backup"
then mkdir $HOME/backup"
fi

for i
do
    if test -f "$i"
    then ln "$i" "$HOME/backup 2>/dev/null ||
           ln "$i" "HOME/backup/$i$$"
         echo -n delete
         rm -i "$i"
    fi
done
```

Fig. 4-3. A modified remove command.

When you execute this program, it first checks for a directory called *backup* in your HOME directory. If *backup* does not exist, this program creates it. It then creates a link between the file that you intend to delete and the *backup* directory. If it can't link the specified file to the *backup* directory because a file by the same name already exists in that directory, then any error messages generated by **ln** are discarded (with the 2>/dev/null error redirection syntax), and the specified file is linked to the *backup* directory by creating a new filename consisting of the file's name concatenated with a random number supplied by the **$$** variable. Finally, the program asks if you want to delete the specified file. If you type "y", the file is deleted.

If you want to recover a file, look in the *backup* directory for that file, then copy it to the correct directory. Periodically, delete all files in the *backup* directory.

If you want all files in the *backup* directory deleted each time you log out from TNIX, you can place the following lines in a file called **bye**, in your personal programs directory:

```
cd $HOME/backup
(nohup rm -fr * .* >/dev/null 2>&1 & ) >/dev/null
logout
```

then type

```
$ . bye
```

instead of

```
$ logout
```

each time you want to log out.

An Example—A Skeleton Shell Program

Figure 4-4 shows an example of a general-purpose shell program. Most shell programs can be written using this program as a starting point. To use this program:

1. Add the flags that you want this program to recognize to the "arglist" variable.
2. Modify the **case** statement so that it recognizes the proper flags for this program. The sample **-f** flag shows how to make this program recognize successive arguments, such as "-f /dev/rhd1". The sample **-t** flag shows how to make one flag stand for several flags.
3. At the end of the figure, where it says "Now execute the program", add the commands that you want this program to execute. The "options" variable contains the list of options that the **case** statement recognized. The "files" variable contains the list of filenames that the **case** statement identified.

```
: <<\!
: Skeleton Shell Program
: Purpose--use as a general shell program.
: If the "debug" variable is set to a non-null value before this program
: is executed, then the "-x" trace execution option is turned on.
: There are three ways to invoke this program with the "-x" option set.
:   1) sh -x program_name arguments_list
:   2) debug=true program_name arguments_list
:   3) debug=true; export debug; program_name argument_list
!
test "$debug" && set -x
: Initialize variables.
: Set up any temporary files, then remove them when this program exits.
temp_file="/tmp/`basename ${HOME}`$$"
: The '[-f] [-t] [-u]' options are only example, and should be changed or
:   removed from the $arg_list variable definition.
files= options= pipe= program_name=`basename $0`
arg_list="[?] [-?] [-] [-f] [-t] [-u] [file1 file2 . . .]"
how_to_use="
    standard input and/or a list of files can be specified:
    cat file1 | $program_name
    $program_name <file1
    cat file1 | $program_name - file2 file3
    $program_name - file2 file3 <file1
    $program_name file1 file2 file3"
: Reset command line arguments--transforms "-ftu" to "-f -t -u"
set - `echo "$@" | sed -e '
    : MARK
    s/\(-[A-Za-z]\)\([A-Za-z]\)/\1 -\2/
    t MARK'`
test "$debug" && set -x
: Read command line arguments, save them in the $command_line
:   variable, initialize the list of options, and
:   initialize the list of filenames.
: The -f, -t, and -u case labels are examples, and should be changed or
:   deleted as necessary.
command_line="$*"
while
      test $# -gt 0
```

Fig. 4-4. A skeleton shell program. (Part 1 of 2)

```
do
      case "$1" in
           -)   pipe=true ;;
           \?)  echo "syntax: $program_name $arg_list $how_to_use"
                exit 0 ;;
           -\?) echo "syntax: $program_name $arg_list $how_to_use"
                exit 0 ;;
           -f)  options="$options -f"
                shift
                options="$options $1"
                continue ;;
           -t)  options="$options -tbl -col" ;;
           -u)  options="$options -T4105" ;;
           -*)  echo "bad option: $1; syntax: $program_name $arg_list"
                exit 1 ;;
           *)   if    test ! -r "$1"
                then
                      echo "file $1 cannot be read!"
                      exit 1
                else
                      files="$files $1"
                fi ;;
      esac
      shift
done
: Reset the command line to its original state
set - $command_line
test "$debug" && set -x
if    test -z "$files" -a -z "$pipe" -a -t 0
then
      echo "No input files!"
      exit 1
fi
trap "
         trap '' 0 1 2 3 13 15
         rm -f $(temp_file)*
         trap 0 1 2 3 13 15
         exit 0
     " 0 1 2 13 15
: If no files are specified, then start writing the standard input to the
:   file named by the $temp_file variable.
if    test "$pipe" = "true" -o ! -t 0
then
      files="$temp_file $files"
      cat > $temp_file <&0-
fi
: Now execute the program.  Replace the following lines with the commands
:   that you want executed.  The following lines show what the
:   $command_line, $options, and $files variables will be set to.
echo the command line used is $command_line
echo the options used are $options
echo the files used are $files
```

Fig. 4-4. A skeleton shell program. (Part 2 of 2)

An Example—A Delay Program

Figure 4-5 shows a shell program that allows delayed execution of a specific command.

```
: Executes command $1 after "sleeping" for $2 hours,
: $3 minutes, and $4 seconds ($3 and $4 are optional).
: Optional flag prefix allows "nice" value, which lowers the execution
: priority of the command.
case $1 in
     -)   nice="nice"
          shift ;;
     -*)  nice="nice $1"
          shift ;;
     *)   nice= ;;
esac
if   test "$#" -lt "2"
then echo "usage: `basename $0` command hours [minutes [seconds]]" >&2
     exit 99
fi
if   test "$2" -gt 17
then echo "can't wait longer than 17 hours" >&2
     exit 1
fi
(    trap "" 1 2 3 15
     sleep `expr $2 \* 3600 + ${3-0} \* 60 + ${4-0}`
     exec $nice sh -c $1
)    2 >&1 &
```

Fig. 4-5. A delay program.

DEBUGGING SHELL PROGRAMS

The first step in debugging a shell program is to check that variables are being set and evaluated properly, and that all program statements, such as **case** and **for**, are complete. In addition, the **echo**, **set –v**, and **set –x** commands are useful for evaluating variables and tracing the execution of a program. The following paragraphs discuss these and other tools such as log files and setting a program's exit status.

Execution Trace

There are two ways to trace the execution of a shell program:

- executing the program as an argument to the **sh –x** command, and
- by placing the **set –x** command at the beginning of the program.

Each command causes the shell to print each command that it executes. For example:

```
$ sh -x compile lister
```

This command line makes the shell print each command in the **compile** program, as it executes that command.

If you want to observe the exact command that the shell executes, after it has expanded the *, [...], and ? special characters, and performed variable and command substitution, set the **-x** shell execution parameter for the current shell by typing:

```
$ set -x
```

This command enables the **-x** execution parameter for the currently executing shell. For example, if you have set the **-x** parameter for the current shell, and you type the **ls *** command, this is what you would see:

```
$ ls *
+ls a.out mover.c mover1.c mover2.c mover3.c
a.out          mover.c        mover1.c       mover2.c       mover3.c
```

To disable the **-x** setting for the current shell, type:

```
$ set -
```

Log Files

Another help in debugging is the log file, which shows the commands executed by a shell program. To create a log file, use the **sh -x** command to execute the shell program, and direct both the standard output and standard error output to a file.

For example, to create a log file that shows all of the commands executed by **compile**, type:

```
$ sh -x compile >logfile 2>&1
```

In the following paragraphs, we'll look at how you can set the exit status of a shell and a C program. (The optional C compiler is supplied with the Native Programming Package.)

Setting the Exit Status of a Program

When you write a program that will be executed by the TNIX shell, you should set the exit status of the program so that useful information about how the program executed and terminated is available to the shell. The shell assigns the exit status value to the **$?** shell variable when the program finishes executing. Generally, the shell's program statements expect a program's exit status to be set to:

0	Tells the shell that the program executed properly, that is, no errors occurred during execution; or
a non-zero value	Tells the shell what sort of error occurred during program execution.

For example, the **diff** program sets the **$?** exit status variable to one of three states:

0	Indicates that there are no differences between the two files that **diff** analyzed.
1	Indicates that there are differences between the two files that **diff** analyzed.
2	Indicates that an error occurred during program execution.

For example, in a shell program, the statement **exit 1** sets the exit status of that program to 1.

In a C program, the statement **exit(1);** sets the exit status of that program to 1. Be sure to place the statement **# include** <**stdio.h**> at the beginning of your C program if you use the **exit** program statement.

A HIGH-LEVEL PROGRAMMING LANGUAGE

The following paragraphs discuss how the shell functions as a high-level programming language. A high-level language typically divides a program into two parts: the *main* part of the program, which oversees and controls execution of other parts of the program, and *subroutines* or *functions* which carry out the actions specified in the *main* part of the program. In the following paragraphs, we will apply this analogy to our discussion of the TNIX shell.

Using TNIX Commands as Subroutines

When you are working with TNIX, you are generally performing a specific task. This task can entail anything from reading your computer mail, playing a computer game, or writing a weekly status report to acquiring data from a laboratory instrument. Here's an analogy between a task that you are performing when you are working in the shell and a task that a program performs.

- When you type one or more command lines in order to perform a task, you are using the shell to direct the execution of your task. Similarly, the program statements in the *main* part of the program direct the execution of a program's task.
- When the shell executes a specific command, such as **ls** or **rm**, that command is usually executed as a subroutine called from the shell—execution control is temporarily transferred to that command. Similarly, statements in the *main* part of a program may call specific subroutines, temporarily transferring execution control to the called subroutine.
- When the command finishes executing, execution control returns to the shell. Similarly, when a subroutine called by the *main* portion of a program finishes executing, execution control is transferred back to the *main* portion of the program.

Because the shell is a program, a new shell program can be executed, or called, from the currently executing shell. Thus, recursive execution of the shell is possible. When you finish working in the shell that you have just executed, program control returns to the calling shell. Each new shell program is known as a subshell, or an "instance" of the shell.

SHELL LANGUAGE REFERENCE SUMMARY

Some commands that you type, such as **newgrp** and **cd**, are part of the TNIX shell. Thus, you will not find a program called **newgrp**, just as you will not find a program called **for**. These commands, and the shell's reserved words, are summarized in the following paragraphs.

break [*n*] Exits from the enclosing **for**, **while**, or **until** loop. If you specify *n*, then exits from the *n*th enclosing loop.

case *variable-name* **in** [*pattern*[¦ *pattern*]...]) *command-list*;;]...**esac**
Executes the first *command-list* where the value of *variable-name* matches the series of characters specified by *pattern*.

cd [*new-directory*] Changes the current directory to the directory specified by the HOME shell variable. If you specify *new-directory*, changes to that directory.

continue [*n*] Resumes execution at the next iteration of the enclosing **for**, **while**, or **until** loop. If you specify *n*, then resumes execution at the *n*th enclosing loop.

do Begins the body of the **for**, **while**, and **until** statements.

done Terminates the **for**, **while**, and **until** statements.

esac Terminates the **case** statement.

eval [*shell-commands*]
Executes *shell-commands* in the current shell, that is, as if you had typed *shell-commands* from the keyboard. Suppose you have a file called *set.vars* that contains the following lines:

```
a=$HOME/projects/laserweapon/source
b=$HOME/projects/laserweapon/object
c=$HOME/projects/laserweapon/documents
export a b c
```

Now, instead of typing the shell commands listed in *set.vars* each time you want to set these variables, you can use the following command line to set these variables:

```
$ eval `cat set.vars`
```

exec [*command*] Executes *command* in place of the current shell. If you type **exec ls** right after you log in, the **ls** command is executed, then you are logged out. If you type **exec ls** while you executing a subshell, the **ls** command is executed, then you are logged out of the subshell.

exit [*n*] Exits from the currently executing shell program, and sets the shell **$?** exit status variable to *n*. If you do not specify *n*, **$?** is set to the exit status of the last program executed within the currently executing shell program.

export [*variable-list*]

Specifies that all shell variables named in *variable-list* are global variables; that is, the values assigned to these variables are available to any programs called by the current shell. Here is an example:

```
$ export TERM TERMCAP IU uP
```

If you do not specify *variable-list*, the names of variables that have been *exported* in the current shell are printed.

fi Terminates the **if** statement.

for *variable-name* [**in** *value-list*;] **do** *command-list*; **done**

This is the shell's **for** statement. The **for** loop executes *command-list* once for each argument in *value-list*.

login [*username*] Logs *username* into TNIX, replacing the current shell with the login shell for *username*. This command is equivalent to typing:

```
$ logout
login: username
```

The **login** command executes the command specified in the seventh field of a user's entry in the */etc/passwd* file when the user logs in. If a command is not specified in the user's entry in the */etc/passwd* file, **/bin/sh** (the TNIX shell) is executed.

newgrp *groupname*

Changes the default group. Your user name must appear in the same line as *groupname* in the */etc/group* file. Since this command is equivalent to typing:

```
$ exec newgrp groupname
```

you will be logged out if you specify an invalid *groupname*. If you want to temporarily log into a different group, then return to your original group, you can type the following commands:

```
$ sh          [work in a new shell]
$ newgrp groupname
              [when you finish working in this new group ...]
$ ^D          [type a CTRL-D]
              [you are now logged into the original group]
```

read [*variable-list*] Reads one line from the standard input, assigning each blank- or tab-separated word to each variable in *variable-list*. If you specify more words in the input line than variables in *variable-list*, the extra words are assigned to the last variable in the list. To see how this works, let's look at an example:

```
$ read a b c d
this is a line of text
$ set
a=this
b=is
c=a
d=line of text
$
```

readonly [*variable-list*]

Variables specified in *variable-list* may not be altered. If you do not specify a *variable-list*, all readonly variable names are printed.

set [**-ceiknpstuvx** [*positional parameters*]]

Sets specific parameters for the current shell. These parameters are valid for both the **set** command and for the **sh** command. If you do not specify *positional parameters* or a specific option, prints the values of all currently set shell variables. Here's an example:

```
$ set
HOME=/usr/lazaruslong
IFS=

PATH=:/usr/lazaruslong/my programs:/bin:/usr/bin:/usr/games
PS1=$
PS2=>
```

To find out which shell parameters are currently set, type:

```
$ echo $-
```

Use *positional parameters* to assign values to the **$1**, **$2**, **$3**, etc. shell positional parameters. For example, if the current directory contains the files *init.c*, *io.c*, and *main.c*, and you want to assign these file names to the **$1**, **$2**, etc. shell positional parameters and then list the values of these parameters, type:

```
$ set *
$ a=0
$ for i
> do
> a=`expr 1 + $a`
> echo '$'$a = $i
> done
$1 = init.c
$2 = io.c
$3 = main.c
```

Here's a short definition of each parameter available with the **set** command:

-c *string* Commands are read from *string*. (The *string* should be enclosed in quotation (") marks if it extends over more than one line.)

-e Exits from the current shell if any command generates a non-zero exit status (**$?** not equal to 0). (You will be logged out if you type **set -e** while in a login shell.)

-i If the **-i** flag is present or if the shell input and output are attached to a terminal, then the currently executing shell is *interactive*. In this case the *terminate* signal is ignored (so that the command

```
$ kill 0
```

does not kill an interactive shell) and the *interrupt* signal (received by the shell when you type a CTRL-C) is caught and ignored.

-k Places all variable assignments that are contained on the same line as a command to be executed in that command's environment. Usually, the variable assignments following a command (on the same line as that command) are passed literally, as parameters, to the command. The **-k** argument allows you to place the variable assignment at any point on a command line. For example, you can execute the **compile** command by typing:

```
$ compile debug=true
```

In this command line, the "debug" variable is set to "true" then passed to the **compile** program, but its value in the current shell is not altered. Compare this to the following command line:

```
$ debug=true; export debug; compile
```

In this command line, the "debug" variable is set to "true" then passed to the **compile** program, and its value in the current shell is not altered.

-n Reads but does not execute commands. Be careful—if you type **set -n,** the shell will not execute any commands that you type—you will have to type a CTRL-D to resume normal operations.

-s If the **-s** flag is present or if no arguments remain, then commands are read from the standard input. Shell output is written to file descriptor 2, the standard error output.

-t Reads and executes one command, then exits. If you type **set -t,** you will be logged out of the shell you are currently executing.

-u Sets the **$?** exit status variable to 1 if you attempt to access a shell variable that has not been set.

-v Displays shell input lines as they are read.

-x Prints both commands and parameters as they are executed. Shell metacharacters are expanded, as appropriate, showing you the actual command that was executed.

- Turns off the **-v** and **-x** options.

sh [**-ceiknpstuvx** [*program-name*] [*program-parameters*]]

Executes

```
program-name program-parameters
```

in a subshell, with the execution parameters for the subshell specified by one or more of the **-ceiknpstuvx** options. The shell's execution parameters are defined in the above description of the **set** command.

shift Renames the shell positional parameters **$2**, **$3**, etc. to **$1**, **$2**, etc.

times Prints the accumulated user and system times for processes run from the shell.

trap [*command-list*] [*signal1*] [*signal2*] ...

If the the first argument to **trap** is a list of one or more commands, then these commands (*command-list*) are executed whenever one of the specified *signals* is received by the shell. (A **signal** is an integer in the range 0-15; these **signals** are listed in Table 4-2, earlier in this section.)

If *command-list* is the null string, the list of **signals** specified in this form of the statement is ignored by the shell. (This is one way to make sure that a CTRL-C or CTRL-D doesn't halt a shell program at the wrong time.)

If *command-list* is not specified, then any *signals* that were previously specified in a **trap** statement are reset to their initial values (when the current shell process was originally invoked).

For example, to make a program ignore the *hangup* signal (1), generated when you type a CTRL-D, and the *interrupt* signal, generated when type a CTRL-C, place the following command at the beginning of your shell program:

```
trap "" 1 2
```

umask [*n*] Specifies the default read/write/execute privileges, in octal, for each new file that you create. For example, to create all files so that only you can read/write/execute them, but no one else can, type:

```
$ umask 077
```

Note that the octal value is the *complement* of the file mode values specified with the **chmod** command. If you do not specify *n*, **umask** displays the current default file creation modes.

until *control-list* ;**do** *command-list*; **done**

Executes the commands in *control-list*, then the *command-list*, until the last command in *control-list* returns a true value (zero exit status).

wait [*process ID*] Waits for one or more processes specified by *process ID* to finish executing. If you do not specify *process ID*, then all currently active child processes (created when the current shell executes a **fork** system call) are *waited* for. The **$?** exit status shell variable is set to the exit status of the process *waited* for.

while *control-list* ;**do** *command-list*; **done**
Executes the commands in *control-list*, then the *command-list*, until the last command in *control-list* returns a false value (non-zero exit status).

TABLES

The following tables summarize the information presented in this section. Table 4-3 shows some commonly used shell variables. Table 4-4 lists the shell metacharacters and reserved words. Table 4-5 summarizes the shell grammar.

Table 4-3
Commonly Used Shell Variables

Variable Name	Description
$#	Number of positional parameters
$0	Name of the program being executed
$1-$*n*	Positional parameters 1, 2, ..., *n*
"$@"	Same as "$1" "$2" "$3" ... "$*n*"
"$*"	Same as "$1 $2 $3 ... $*n*"
$?	Exit status of the last-executed program
$!	Decimal process number of last-executed background command
$-	Currently set shell flags
$$	Decimal process number of the currently executing shell
HOME	The HOME directory, also the default argument to **cd**
IFS	Internal field separator—characters used to delimit shell arguments
IU	HSI I/O port currently connected to an 8540 IU or 8550 MDL
KSH	Default command line options for the TEKTRONIX Keyshell interface
MAIL	Pathname of the directory that incoming mail is stored in
MORE	Default command line options for the **more** command
PATH	List of directories to search for executable commands
PS1	Primary shell prompt, usually set to "$ "
PS2	Primary shell prompt, usually set to "> "
TERM	Terminal-type variable, describes the capabilities of the terminal that you are using
TERMCAP	Pathname for the terminal description database file. The value of the TERM variable specifies an entry in this database file
uP	Current target microprocessor

Table 4-4
Shell Metacharacters and Reserved Words

Type	Syntax	Description
SYNTACTIC SYMBOLS	|	Pipe symbol—connects the output of one command to the input of another command
	&&	Execute following command if preceding command returns *true*, that is, sets the **$?** exit status variable to zero
	||	Execute following command if preceding command returns *false*, that is, sets the **$?** exit status variable to zero
	;	Command separator—separates commands typed on the same line
	;;	**case** delimiter—terminates a case label and associated command list in a **case** statement statement
	&	Background commands—executes the preceding command or parenthesis-enclosed command list concurrently
	(...)	Command grouping—parenthesis-enclosed commands are executed in a separate shell process
	<	Redirect input from a file or file descriptor
	<<	Redirect input from the following text
	>	Redirect output to a file or file descriptor
	>>	Redirect output, appending to an existing file
PATTERNS	*	Match any character, including the null character, other than the slash "/" character
	?	Match any single character
	[...]	Match any one of the enclosed characters or range of characters
SUBSTITUTION	${...}	Substitute shell variable
	`...`	Substitute command output
QUOTING	\	Overrides the shell's interpretation of the next character
	'...'	Overrides the shell's interpretation of the enclosed characters, except for the apostrophe (') character
	"..."	Overrides the shell's interpretation of the enclosed characters except for the quotation (") mark, dollar sign ($), accent grave (`), and backslash (\) characters
RESERVED WORDS		if then else elif fi case in esac for while until do done { }

Table 4-5
Shell Grammar

Type	Defined As
item	word input-output name=value
word	a sequence of non-blank characters
name	a sequence of letters, digits, or underscores starting with a letter or underscore character
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**
pipe	command pipe ¦ command
andor	pipe andor **&&** pipe andor ¦¦ pipe
command-list	andor command-list command-list**&** command-list**;** andor command-list**&** andor
input-output	>file <file >>file <<word
file	word **&**digit **&–**
case-part	pattern**)** command-list**;;**
pattern	word
	pattern ¦ word

Table 4-5 (Cont)
Shell Grammar

Type	Defined As
else-part	**elif** command-list **then** command-list else-part **else** command-list empty
empty	
digit	0 1 2 3 4 5 6 7 8 9

Section 5
THE TNIX EDITOR

TABLES

Section 5

THE TNIX EDITOR

INTRODUCTION

The TNIX text editor **ed** is a line-oriented editor that allows you to create any text—program code, command files, or correspondence. Usually, you will enter the text, correct or modify it, and store it in a file for immediate or future use.

This section presents the following topics:

- **Basic tasks.** Contains the few **ed** commands with which you can accomplish most editing tasks.
- **Advanced topics.** Contains more **ed** commands, plus further explanations of features such as line addressing and context searching. Also describes **ed** invocation in detail.
- **Reference summary.** Contains tables that summarize **ed** commands and features.

BASIC TASKS

Most editing tasks can be accomplished using only a few **ed** commands. This subsection shows how to invoke the editor, manipulate text and files, and exit from **ed**, using a few basic commands. Additional **ed** commands can increase your speed and productivity while editing, and are presented in the following subsection, "Advanced Topics".

Starting the Editor

To edit your file with **ed**, enter:

```
$ ed yourfile
```

(For information about other ways of invoking **ed**, see the discussion "Invocation", in the following subsection, "Advanced Topics".

If this is the first time you've edited *yourfile*, **ed** responds with a question mark and the filename:

```
$ ed yourfile
?yourfile
```

The question mark is **ed**'s feedback, which you will receive whenever **ed** can't open a file or finish a command (in this case, *yourfile* is new, so **ed** couldn't open it).

If *yourfile* already exists, **ed** responds with the number of characters in the file:

```
$ ed yourfile
216
```

After you have entered **ed**, **ed** silently waits for commands. **Ed** commands are entered at the beginning of a line, usually one command per line. (The **p**, **l**, **g**, and **v** commands may appear on a line with another command; these commands are explained later.)

Once you've entered **ed**, all the editing you do affects only a copy of the file you're working on. This copy is known as the editing "buffer". **Ed** also remembers the last file given to the **ed** command as the "current file".

No changes are made to the file you're working on until you enter a **w** (write) command. When you do want to save editing changes, issue a **w**. At that point, the contents of the editing buffer are sent to the current file, and the actual file is changed.

Set the Prompt Character. Ed does not display a prompt character unless you define one. You can have **ed** prompt you for commands (which helps you identify the insert and command modes more easily) by defining a prompt character with the **P** command. The prompt character appears whenever **ed** is in command mode. There are two ways to set a prompt character: you can enter the **-p** option when invoking ed, and the prompt character is an asterisk. For example:

```
$ ed -p testfile
385
*
```

You can establish a different **ed** prompt character by using the **P** command. For example, to set the prompt character to >, enter

```
P>
```

Manipulating Text

The following pages describe the commands most often used to manipulate text during an average editing session—commands for adding text, deleting text, substituting, and moving around in the file. The basic commands for manipulating text are:

`n`	**address of a line—placed before commands**
`i`	**insert**
`a`	**append**
`d`	**delete**
`p`	**print**
`/text/`	**search for text**
`s`	**substitute**
`u`	**undo**

Address a Line. Ed is a line-oriented editor; that is, for any given command, **ed** has to know which line to operate on. You can find out what line you're on by entering a period and an equals sign:

```
.=
```

(**Ed** does not routinely display line numbers).

You can specify lines in four basic ways:

1. *By entering nothing* before the command. The command then affects the "current line" (last line operated on by an **ed** command). You can find out the number of the current line at any time by typing a period and an equals sign.
2. *By entering a number* before the command. The command then affects the line in the file with that number. You may also enter two numbers with a comma in between. The command then affects the range of lines from the first number to the second number, inclusive.
3. *By entering a search command.* The command following the search then affects the next line in the file that contains the search item. The item may contain a sequence of literal characters or a sequence of special characters that define a more general pattern. (See the discussion under "Advanced Topics" for more information on searching and search patterns).
4. *By entering special address characters* which **ed** recognizes as addresses. The address characters are:

`.`	**(period) the current line**
`+`	**the next line in the file**
`-`	**the previous line in the file**
`$`	**the last line in the file**
`1,$`	**all lines in the file (first to last, inclusive)**

   Typing RETURN prints the next line and is a handy way to step through the buffer. Typing a "-" prints the previous line, and can step you backward through the buffer.

The following are some examples of giving an address to the **d** (delete) command:

`d`	**deletes the current line**
`3d`	**deletes line number 3**
`3,58d`	**deletes lines 3 through 58, inclusive**
`/Now is/d`	**deletes the next line in the file that contains the words "Now is"**

In this section, the letter *n* represents an addressed line in the file, whether the address is derived from no specified address, a number, a range of numbers, a search string, or a special character.

For more information on addressing, refer to the heading "Addressing", under "Advanced Topics".

Add Text (Insert and Append). The **i** command (insert) adds new text above the line. For example, to add text above the current line, enter:

```
i
Now is the time
for all good
.  (period)
```

Your file will now contain the text you typed, but not the initial **i** command or the final period.

Similarly, the **a** command (append) adds new text below the addressed line. For example, the following command sequence adds the new text "if not why not" after line 45 in the file:

```
45a
if not why not
.
```

Both the **i** and **a** commands place you in "insert mode" while you are adding text. You may leave insert mode and return to command mode by entering a period at the beginning of a new line.

Delete Text. The **d** command deletes the addressed line or lines from the file. The following are some examples of the **d** command:

`d`	**deletes the current line**
`3d`	**deletes line number 3**
`3,58d`	**deletes lines 3 through 58, inclusive**

Print Text. The **p** command displays the addressed lines on the terminal. The following are some examples of the **p** command:

`p`	**prints the current line**
`1,6p`	**prints the first six lines in the file**
`1,$p`	**prints all the lines in the file**

Search for a String. If you know the word or phrase you want to edit, but you don't know the number of the line that contains that word or phrase, you can find the line by using a search command (also known as "context searching"). For example, the following command searches for and prints the next line that contains the word "speling":

```
/speling/
features could include a speling checker, or other tools
```

If "speling" is found, **ed** prints the line in which it was found (but not the line number), and that line becomes the current line.

Forward searches start at the line after the current line, proceed to the end of the file, wrap around to line 1, then proceed to the current line before giving up. If the item is not found, **ed** prints its question mark for feedback.

You can search for any combination of letters, numbers, or other characters. You may also use a search wherever you would use any other address, in front of a command. The command would then operate on the line found by the search, if any. The following are examples of using search strings as addresses:

`/{mainline/`	**finds and prints line containing "{mainline"**
`/1040/d`	**deletes line containing the number "1040"**

You may also form search strings that match patterns of characters, not just literal characters. Such a string, containing special pattern-matching characters, is known as a "regular expression". For more on regular expressions, refer to "Advanced Topics" in this section.

Substitute Text. The **s** command (substitute) allows you to change individual characters within a line or a group of lines. This way, you can correct spelling errors or typing mistakes without reentering the entire line.

For example, assume the current line contains

```
features could include a speling checker, or other tools
```

The following sequence corrects the misspelled word and prints the new current line:

```
s/speling/spelling/
p
features could include a spelling checker, or other tools
```

You may also append the **p** command to the **s** command:

```
s/speling/spelling/p
features could include a spelling checker, or other tools
```

The general form of the substitute command is

```
ns/search/replace/
```

where *n* is any addressed line or range of lines, *search* is the string of characters to be discarded, and *replace* is the string of characters to be used instead. If you omit the address, the current line is assumed. If you omit the *search* string, **ed** uses the most recent search string:

```
/speling/          searches for next line containing "speling"
s//misspelling/p   substitutes "misspelling" for remembered "speling"
```

If you give a range of addresses to the **s** command, the command will make the substitution on the first occurrence of the search item in each addressed line. (To make a substitution for every occurrence in the line, see the **g** (global) command).

The following are examples using the **s** command:

```
Nowxx is the       original line
s/xx//             deletes the two extra "x"s
32s/hte/the/       changes "hte" to "the" in line 32
1,$s/file-name/filename/
                   changes the first occurrence of "file-name" in each
                     line of the file to "filename"
```

Undo. The **u** command (undo) restores the current line to the state it was in before the last substitution. The following sequence shows an example:

```
p
features will be available at the first release
s/will/will not/p
features will not be available at the first release
u
p
features will be available at the first release
```

Undo works only if the last line substituted is the current line.

Manipulating Files

The basic commands for manipulating files are:

`e`	**edit another file**
`r`	**read the contents of a file into the file being edited**
`w`	**write the contents of file being edited to another file**

Edit Another File. Once you are already in **ed**, you can use the **e** command to bring in another file for editing.

CAUTION

If you have text already in the buffer, it will be lost by the ***e*** *command. If you want to save the text already being edited, make sure you write it* ***(w*** *command) into a file before using the* ***e*** *command.*

For example, the following command brings file *test2* into the buffer for editing:

```
e test2
```

test2 then becomes the current file. (You can find the name of the current file at any time by entering the **f** command).

Read Text from Another File. The **r** command reads the contents of a file into the editing buffer. An address may be given to the **r** command, in which case the new file is read in below the given address. With the **e** command, the old contents of the buffer are not lost or overwritten. The following are examples of using the **r** command:

`r myfile`	**adds contents of** *myfile* **at the end of the file being edited**
`32r bplate`	**adds contents of** *bplate* **after line 32**
`/here/r txt`	**adds contents of** *txt* **after next line that contains the word "here"**

Write to a File. The **w** command writes the contents of the editing buffer to the named file. An address or range of addresses may be given to the **w** command. If a single address is given, that line is written. If a range of addresses is given, all lines in the range are written.

The following are examples of using the **w** command:

`w`	**writes all lines in buffer to the current file**
`w myfile`	**writes all lines in buffer to the file** *myfile*
`1,32w bplate`	**writes lines 1–32 to the file** *bplate*

After writing out the specified lines, **ed** responds with the number of characters written.

Exiting from the Editor

To end a typical editing session, you save the results of your editing (write to the current file) and exit from the editor. The commands you use when quitting the editor are:

```
w          write (save) to current file (optional)
q          exit from ed
```

You use both **w** and **q**, in that order, to save the results of your edit session and return to TNIX.

If you haven't issued a **w** command since your last editing command and you try to quit, **ed** responds with a question mark to remind you to write first. If you don't want to save the changes you've made, enter **q** again to exit from the editor (**ed** only reminds once).

Summary of Basic Tasks

In summary, a basic editing session takes the following form:

```
$ ed myfile
[ number of characters in myfile ]
.
.
.   [ editing commands ]
.
.

w
[ number of characters written ]
q
$
```

Table 5-1 summarizes the **ed** commands used to perform basic editing tasks.

Table 5-1
Ed Basic Editing Tasks

Task	Command Usage	Result
Invoking Ed	**ed** *file*	Invokes **ed** on *file*.
	P[*x*]	Sets prompt character to *x*.
Manipulating Text	*n command*	Addresses a line—**ed** *command* is done on line *n*. *n* can be nothing (the current line), a single line number (3), a range of line numbers (3,58), or a search string (*/string/*).
	n **i** *text...* .	Inserts *text* before line addressed by *n*.
	n **a** *text...* .	Appends *text* after line addressed by *n*.
	n **d**	Deletes line(s) addressed by *n*.
	n **p**	Prints line(s) addressed by *n*.
	/string/	Searches for next occurrence of *string* and prints line. *String* may be a regular expression.
	n **s**/*search*/*replace*/	Substitutes *replace* for first *search* in lines addressed by *n*.
	u	Undoes last substitution, if that line is current line.
Manipulating Files	**e** *file*	Brings *file* into the buffer for editing. Destroys previous contents of buffer (if any).
	n **r** *file*	Appends *file* after line addressed by *n*.
	n **w** *file*	Writes lines addressed by *n* to *file*.
Exiting from Ed	**w**	Writes to current file (saves changes).
	q	Exits from **ed**.

The previous discussion showed you the basic **ed** commands to accomplish ordinary tasks. The following subsection presents information you will need to make maximum use of **ed**'s capabilities.

ADVANCED TOPICS

This subsection presents the following topics:

- additional **ed** commands, error messages, and interrupted commands.
- global commands (commands that operate on the entire file).
- more information on addressing.
- a discussion of regular expressions and their uses.
- a description of optional ways to invoke **ed**.
- software tools that support **ed**.

More Ed Commands

The following paragraphs briefly explain **ed** commands that were not included in the "Basic Tasks" discussion. Each command entry includes a descriptive phrase and examples. For more detailed information on all **ed** commands, refer to Table 5-3 at the end of this section, or type **man ed** for online information that describes the editor.

This subsection discusses the following **ed** commands:

`c`	**change text**
`m`	**move text**
`t`	**transfer (copy) text**
`j`	**join lines**
`k`	**mark a line, for addressing that line**
`l`	**list (print) lines, including characters that are normally not displayed**
`?`	**search backward**
`//`	**repeat last forward search**
`??`	**repeat last backward search**
`!`	**execute a single TNIX command**
`!sh`	**execute a sequence of TNIX commands (enter CTRL-D to return to ed)**

Change Text. The **c** command changes the text of the addressed lines. Like the **i** and **a** commands, **c** puts you in insert mode while you enter new text, and you terminate insert mode by entering a period at the beginning of a new line. The following are examples of using the **c** command:

```
c                   changes current line to "Now"
Now
.
9,/now/c            changes all lines from line 9 to the next line
if not              that contains the word "now", to the text "if not"
.
+c                  changes next line to "how now"
how now
.
```

Move Text. The **m** command moves the addressed lines to a new place in the file. Some examples are:

`m32`	**moves current line to right after line 32**
`5,25m200`	**moves lines 5–25, inclusive, to right after line 200**

Copy Text. The **t** command copies (transfers) the addressed lines to a new place in the file. Some examples are:

`t32`	**copies current line to right after line 32**
`5,25t200`	**copies lines 5–25, inclusive, to right after line 200**

Join Lines. The **j** command joins the addressed lines (removes newlines). Some examples are:

`j`	**joins current line with next line**
`-,.j`	**joins previous line and current line**
`1,3j`	**joins lines 1–3 (removes two newlines)**

Ed also lets you split a single line into two or more shorter lines by "substituting a newline". For example, to break a line between "x" and "y", enter:

```
s/xy/x\
y/
```

The "\" at the end of the first line makes the newline no longer special. That is, **ed** no longer takes the newline as signaling the end of the **s** command, but reads it as a literal newline to be included in the substituted text.

Mark Text. The **k** command puts an invisible mark at the specified line, so you can later address that line by its mark name. Marks are useful for "cutting and pasting" tasks—they allow you to move or copy blocks of text, without having to keep track of specific line numbers. A mark remains associated with a line only while you are in the editor, or until you delete the line or mark a different line with that mark name. Mark names are single characters.

Some examples of setting marks with the **k** command:

`ka`	**marks current line with mark a**
`/Major/kb`	**marks next line containing "Major" with mark b**
`100ka`	**marks line 100 with mark a**
`200kb`	**marks line 200 with mark b**
`300kc`	**marks line 300 with mark c**

Some examples of using marks are:

`'a`	**goes to mark a**
`'a,'bp`	**prints lines from mark a to mark b**
`'a,'bm$`	**moves lines from mark a to mark b to the end of the file**
`.,'at0`	**copies (transfers) lines from current line (period) to mark a, to the beginning of the file**
`'a,'bt'c`	**copies (transfers) lines from mark a to mark b after mark c**

Print Text. The **l** command displays the contents of the editing buffer, including characters that are normally non-printing, such as control characters. Any characters that are normally non-printing are represented by a backslash followed by the octal representation of that character. For example:

```
p                 if the p command results in just a bell ring,
(rings a bell)      the line probably contains a bell character
l                 shows the octal representation of the "bell" character
\07
```

Search Backward. In addition to forward searches, **ed** can also search backwards in the file. For example:

```
?Somehow?         searches for previous occurrence of "Somehow"
                    (next occurrence, backward)
```

Repeat Searches. Ed provides several shorthand notations for repeating searches, so you don't have to reenter the search command:

```
//                repeats last forward search
??                repeats last backward search
```

Semicolon. In **ed**, the semicolon ";" is used to separate two searches in an address, when two search strings are used to specify a range of lines. (If you use a comma, each search string prints its line, and the first found line is not remembered.) For example:

```
/ab/;/bc/p        prints the range of lines from "ab" to "bc"
/thing/;//        finds and prints the second occurrence of "thing"
0;/thing/         finds and prints the first occurrence of "thing"
```

Executing Other TNIX Commands. Sometimes you may want to execute one or more TNIX commands without leaving **ed**. You may do this with the **!** and **!sh** commands, also known as "escaping to the shell":

```
!command          executes command, then returns to ed
!sh               executes TNIX commands until CTRL-D is entered,
                    then returns to ed
```

When the single shell command (**!**) is finished, or when you enter CTRL-D to get out of the new shell (**!sh**), **ed** prints an exclamation point to tell you it is ready to accept more **ed** commands.

The following are some examples of using "shell escapes":

```
!date                     shows current date, then returns to ed
Wed Aug 30 ...
!
!mail joeb                sends a quick note to joeb., then returns
Break any time now.         to ed
<CTRL-D>
!
!sh                       copies and removes some files, then returns
$ cp file1 file2            to ed
$ rm temp*
$ <CTRL-D> !
```

Error Messages

Ed's message, when it doesn't understand or cannot finish a command, is

```
?
```

With some exceptions, you cannot receive any more error information, but once you are familiar with the editor, errors become obvious as soon as the question mark appears. Here are some situations where you might receive a question mark:

after a search	**ed** couldn't find the object being searched for
after an **e** command	**ed** couldn't open the file—it may be new, or you may not have access to it
after a **q** (quit) command	you have not done a **w** since your last command
after any command	command may be misspelled or have wrong syntax

Interrupting the Editor

If you enter CTRL-C while **ed** is executing a command, the command is interrupted and **ed** restores the file as much as possible to what it was before the command began. However, some changes are irrevocable—if you are reading or writing a file or making substitutions or deleting lines, these will be stopped in some clean but unpredictable state. The current line may or may not be changed.

In the case of the **p** command, the current line is not changed until the printing is done. For example, if you enter the command to print the whole file on the screen

```
1,$p
```

and enter CTRL-C when you see an interesting line, that line is **not** necessarily the new current line. The current line is left where it was when the **p** command got underway, because the **p** command was interrupted before it could finish.

Global Commands

The **g** command (global) executes one or more **ed** commands on all lines in the current file that match a given string. For example:

```
g/peling/p                  prints all lines that contain "peling"
g/peling/s//pelling/gp      substitutes for every occurrence of "peling"
                              on every line, then prints each corrected line
```

The initial "g" in a global search means "search throughout the entire file for the following string, but operate only on the first instance on each line". The concluding "g" means "operate on each instance on a line". The concluding "p" prints every changed line when an initial "g" is used, but prints only the last changed line if "1,$" is used.

The **v** command is similar to "g", except that commands execute on every line that does *not* match the string following **v**.

The following are examples of global commands:

`/string/`	**searches for next occurrence of "string" and prints the line**
`g/string/p`	**searches for and prints every line that contains "string"**
`v/string/p`	**searches for and prints every line that does not contain "string"**
`g/strl/s//str2/p`	**searches for every line that contains "str1", substitutes "str2" for first occurrence of "str1" on each line, and prints each altered line**
`g/strl/s//str2/gp`	**same as preceding, except substitutes "str2" for every occurrence of "str1" on each line**
`1,$s/strl/str2/gp`	**same as preceding, except prints only the last line substituted. Issues error message if "str1" not found**

Multiline Global Commands. Sometimes you want to globally execute more than one command. In that case, enter each command on its own line, with a backslash at the end of each line except the last. For example, to change "x" to "y" and "a" to "b" on all lines that contain the word "thing", enter:

```
g/thing/s/x/y/ \
s/a/b/
```

It is also possible to execute **a**, **c**, and **i** commands under a global command; as with other multi-line constructions, all you need to do is to add a backslash at the end of each line except the last.

More on Addressing

Address Arithmetic. You can save typing when specifying lines by using "-" and "+" alone as line numbers. "-" is a command to move up one line in the file, while "+" moves down (forward) one line. You can string several minus (or plus) signs together to move backward (or forward) that many lines. In addition, "+" and "-" can be combined with searches using "/.../" and "?...?", and with "$".

Some examples:

`---`	**moves up three lines**
`-3`	**moves up three lines**
`-,.s/bad/good/`	**changes "bad" to "good" on the previous line and on the current line**
`/thing/--`	**finds the line containing "thing", and positions you two lines before it**

Current Line. You may want to know how a command will affect the current line without actually executing the command. For a complete listing of how a command will affect the current line, see Table 5-2.

Table 5-2
Ed Commands in Relation to the Current Line

Command		Sets Current Line To
a	append	last line input; if no input, last addressed line
c	change	last line input; if no input, line after last line deleted, or $
d	delete	line after last line deleted, or $
e	edit	last line of buffer
i	insert	last line input; if no input, line before addressed line
j	join	resulting line
l	list	last line listed
m	move	new location of last line moved
p	print	last line printed
r	read	last line read
/str/	search	last line matching search *str*
s	substitute	last line substituted
t	transfer	last line of copy
u	undo	unchanged
w	write	unchanged
!	escape to shell	unchanged

Regular Expressions

When searching for items within your file, you may form search strings that will match patterns of characters, not just literal characters. Such a string, containing special pattern-matching characters, is known as a "regular expression". Regular expressions are used in other software tools besides **ed**, such as **grep** and **sed**.

The following is a list of the special characters used in forming regular expressions:

^	**Matches the search item only if the item occurs at the beginning of a line**
$	**Matches the search item only if the item occurs at the end of a line**
.	**(period) Matches any single character**
*	**Matches 0 or more characters (repetition character)**
[ ]	**Matches any one of the characters within the brackets**
&	**Used only on the right side of a substitution, where it means "whatever was matched on the left side"**
\(\)	**Encloses or tags part of the search item for later use in the replacement item**
\1	**Used only on the right side of a substitution, where it means "item enclosed in first pair of \(\) in left side". \2, \3, etc. are also available**

The backslash "\" removes the special significance of these pattern-matching characters (including a backslash itself). If you need to use one of the special characters in a substitute command, you can remove its special meaning by preceding it with the backslash. For example, to change a group of three periods to a semicolon, enter:

```
s/\.\.\./;/
```

If BACKSPACE and CTRL-U are the character-erase and line-kill characters on your terminal, you have to type a backslash before entering a BACKSPACE or CTRL-U in commands to **ed** if you want the BACKSPACE and CTRL-U characters to be taken literally.

NOTE

*When you are adding text with **a** or **i** or **c** (that is, when you are in insert mode), **ed** accepts the special characters literally. You should not combine them with backslashes unless you really want the backslashes to appear in your text.*

The following paragraphs briefly discuss each of the special characters used in regular expressions. Table 5-4 at the end of this section summarizes special characters in regular expressions.

^ (match at beginning of line). The caret in a search string tells **ed** to find the following string only if it occurs at the beginning of a line. Used this way, the caret must be the first character of the regular expression.

For example:

`/^means/`	**finds "means" only if "means" occurs at beginning of line**
`/means^/`	**finds literal string "means^"**

$ (match at end of line). The dollar sign in a search string for a substitute command tells **ed** to find the string only if it occurs at the end of a line. Used this way, the dollar sign must be the last character of the regular expression.

`/means$/`	**matches "means" only if "means" occurs at end of line**
`/$means/`	**matches literal string "$means"**

. (match any character). A period in a search string matches any single character. If you want to search for a literal period, precede it with a backslash. For example:

`/./`	**matches any character**
`/\./`	**matches a literal period**

*** (match zero or more characters).** The asterisk in a search string matches zero or more instances of the character immediately before the asterisk. Again, if you want to search for a literal asterisk, precede it with a backslash. For example:

`/.*/`	**matches any characters at all, no matter how many**
`/x-*/`	**matches zero or more minus signs after an x**
`/\*/`	**matches a literal asterisk**
`/*/`	**won't match anything—improper syntax for a search item**

Because the asterisk and its preceding character allow **ed** to go ahead and operate even on zero matches, some substitutions can have unexpected results. For example:

`abc`	**original line**
`s/x*/-/p`	**matches zero occurrences of x as first character**
`-abc`	**in the line, so a minus is substituted for the first zero instance**
`s/x*/-/gp`	**matches zero occurrences for every character in the line,**
`-a-b-c-`	**so a minus is substituted for every zero instance**

[ ] (match characters in a class). The square brackets tell **ed** to match a character if it is part of the class of characters defined within the brackets. Any characters can appear within the square brackets, and none is special. Even the backslash loses its significance.

The following are examples of using classes of characters:

`/[0-9]/`	**matches a single digit**
`/[0-9]*/`	**matches zero or more digits**
`/[a-z]/`	**matches a single lowercase letter**
`/[a-z]*/`	**matches zero or more lowercase letters**
`/[A-Z]/`	**matches a single uppercase letter**
`/[.[]/`	**matches a period or a left bracket**
`/[-+\*\/]`	**matches any one of arithmetic symbols - + * /**

To include a "]" within the brackets, make it the first character after the left bracket. To include a hyphen "-" in brackets, make it either the first or last character.

An initial caret within square brackets means "all characters except the following characters". For example,

```
[^0-9]
```

stands for "any character *except* a digit".

\(\) (tag part of a search item). You can use the backslash-parentheses within a search item to "tag" the enclosed part. Later, the tagged pieces can be used as replacement parts.

In the left side of a substitution, \(and \) surround the matched item to be tagged. In the right side of a substitution, \1 refers to the first tagged item, \2 refers to the second tagged item, and so on.

The following example converts a text file containing both first and last names to a file containing last names only, using a global search and substitution with regular expression characters (including backslash-parentheses).

```
Doe, John                  original lines
Queue, Suzy
g/^\([^,]*\).*/s//\1/p     changes original lines to last names only
Doe
Queue
```

Here is a breakdown of the global search and substitution, presenting the characters of the global command in the order in which they appear:

```
g/              for each line in the file, search for,
^               at the beginning of a line,
\(              and tag,
[^,]*           a string of zero or more characters that are not commas (last names)
\)              end the tag
.*              match the rest of the line
/               end the global search item
s//             substitute for that entire searched item (original line)
\1              the tagged piece (last name only)
/               end the substitution
p               and print the changed line
```

Invocation

The usual way to invoke **ed** is to enter an **ed** command with a file to edit:

```
$ ed intro.ms
?
```

However, there are more ways to invoke **ed**. The following paragraphs discuss **ed**'s invocation options.

Syntax for Ed Command

The formal syntax for the **ed** command is:

ed [**-cpx**] [*file*]

-c Suppresses the printing of character counts by the **ed** commands **e** (edit), **r** (read), and **w** (write).

-p Causes **ed** to turn on its prompt character (*).

-x Used only with the **x** command of the optional 8560 Series MUSDU Native Programming Package. If **-x** is present, an *x* command is simulated first to handle an encrypted file.

Examples

```
$ ed -p main.ps          Invokes ed on file main.ps, turning
25546                      on the asterisk as the prompt character.
*

$ ed -cp main.ps         Invokes ed on file main.ps,
*                          suppressing the printing of the number of
                           characters in main.ps.

$ ed -c main.ps          Invokes ed on file main.ps,
                           suppressing the printing of the number of
                           characters in main.ps,
                           and not displaying a prompt character.
                           Ed silently waits for commands.
```

Supporting Tools

TNIX offers several tools and techniques that support the editor. Once you know **ed**, these tools are easy to use, because they are all based on the editor. The following paragraphs contain some examples of these tools.

Grep

Sometimes you want to find all occurrences of a word or pattern in a set of files. To edit each file separately can become time-consuming, and may be impossible because of size limits in **ed**.

The **grep** command provides a tool for this task. "grep" comes from the generic **ed** global command:

```
g/re/p          (global/regular expression/print)
```

Grep searches through a set of files and prints every line that contains the specified pattern. The following example prints lines containing the word "thing" in *file1*, *file2*, etc:

```
$ grep 'thing' file1 file2 file3 file4
```

Grep also shows the filename in which the line was found, so you can edit it later.

The **grep** search pattern can be any pattern permissible in **ed**, because **grep** and **ed** use the same mechanism for pattern searching. It is wise to enclose the pattern in the single quotes '...', because some characters also mean something special to the TNIX shell. If you do not quote the regular expression, the shell may try to interpret the characters as TNIX commands before **grep** can process them.

To find lines that don't contain a pattern, use the **-v** option of **grep**. For example, to find all lines that do not contain "thing", enter:

```
$ grep -v 'thing' file1 file2
```

As another example, the following command displays all lines that contain "x" but not "y":

```
$ grep x file... | grep -v y
```

Grep has many options; for additional information, type **man grep**.

Editing Scripts

If you must perform repeated editing operations on a set of files, the easiest way is to make up a "script", a file that contains the operations you want to perform. You can then apply this script to each file in turn. For example, suppose you want to change every "file" to "FILE", and every "Tnix" to "TNIX" in a large number of files. Then put into a file called "script", the lines

```
g/file/s//FILE/g
g/Tnix/s//TNIX/g
w
q
```

There are two ways to invoke **ed** on the files using the commands in the "script":

```
$ ed file1 < script
$ ed file2 < script
....
```

or

```
$ for i in file1 file2
> do ed $i <script
> done
$
```

Sed

Sed ("stream editor") is a version of the editor with restricted capabilities but which can process input files of unlimited size. **Sed** copies its input to its output, applying one or more editing commands to each line of input. It handles input too large for **ed** , and performs conditional testing and branching.

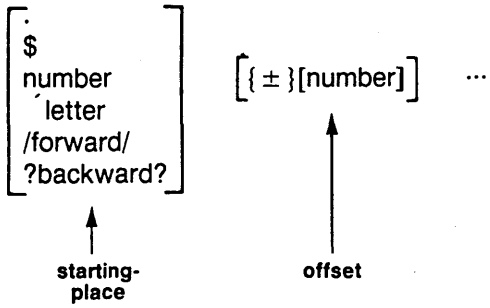
Sed is available only in the optional 8560 Series MUSDU Native Programming Package. For more information, refer to the Native Programming Package User's Manual.

ED REFERENCE SUMMARY

Table 5-3 summarizes each **ed** command. The tables in this subsection use the following notation:

Boldface	characters are entered literally.
Italic	characters represent parameters for you to enter.
Brackets [ ]	surround optional parts of the command.
n	represents an address: a line, range of lines, search string, mark, or special address character.

Line addresses represented by *n* have the following formal syntax:



In the absence of a *starting-place,* "the current line" is assumed. In the absence of an *offset,* 1 is assumed. For example:

```
?here?-2
```

addresses the second line before the first occurrence of "here", searching backwards.

Table 5-4 summarizes searches and regular expressions in **ed.**

As additional reference, you will find an alphabetical list of **ed** commands in the online manual page by entering the command **man ed.**

Table 5-3
Ed Command Quick Reference

Task	Command and Usage		Result
Create Text	**a**	*n*a *text* .	Appends *text* below line *n*.
	i	*n*i *text* .	Inserts *text* above line *n*.
Change Text	**c**	*n*c *text* .	Changes *text* of line(s) *n*.
	d	*n*d	Deletes line(s) *n*.
	s	*n*s/*str1*/*str2*/	Substitutes *str2* for *str1* in line(s) addressed by *n*.
	u	u	Undoes substitution on last line substituted, if that line is current line.
Search	/	/*string*/	Searches forward for next occurrence of *string* in file, prints line if found.
	//	//	Repeats last forward search.
	?	**?***string***?**	Searches backward for previous occurrence of *string* in file, prints line if found.
	??	**??**	Repeats last backward search.
Move Text	**m**	*n*m*d*	Moves (appends) line(s) addressed by *n* to destination line *d*.
	t	*n*t*d*	Transfers (copies and appends) line(s) to destination line *d*.
	j	*n*j	Joins addressed range of lines or, if no lines are addressed, joins current line with following line.
Print Text	**l**	*n*l	Lists addressed lines.
	p	*n*p	Prints addressed lines.

Table 5-3 (Cont)
Ed Command Quick Reference

Task	Command and Usage		Result
Manipulate Files	**e**	e [*file*]	Edits a new file—loses old contents of buffer.
	E	E [*file*]	Edits a new *file*—loses old contents of buffer. No warning if you haven't written (saved) old contents first.
	f	f	Prints current filename.
	f	f *file*	Changes current filename to *file*.
	r	*n*r [*file*]	Reads (appends) *file* at addressed line.
	w	*n*w [*file*]	Writes addressed lines to *file*. Default is entire contents of buffer.
	W	*n*W [*file*]	Writes (appends) addressed lines to *file*. Default is entire contents of buffer.
Miscellaneous	**P**	P*x*	Sets prompt character to *x*.
	k	*n*k*a*	Marks addressed line with symbol *a*. This line now may be addressed as *'a*.
	g	*n*g/*expr*/*cmds*	Global commands: within range of lines addressed by *n*, executes *cmds* on each line containing *expr*. *cmds* are any **ed** commands except **g** or **v**. *expr* is any regular expression.
	v	*n*v/*expr*/*cmds*	Global commands: within range of lines addressed by *n*, executes *cmds* on each line **not** containing *expr*. *cmds* are any **ed** commands except **g** or **v**. *expr* is any regular expression.
	x	x*key*	Uses *key* to encrypt and decrypt files when using **w**, **r**, and **e** commands. Used only with the **crypt** command of the Native Programming Package.
	=	*n*=	Returns number of addressed line.
	!	!*command*	Escapes to TNIX shell and executes *command*.
	!sh	!sh	Escapes to a new TNIX shell and executes TNIX commands until CTRL-D is entered.
Exit from **Ed**	**q**	q	Exits from **ed**.
	Q	Q	Exits from **ed**. No warning if you haven't written (saved) first.

Table 5-4
Search and Regular Expression Quick-Reference

Command		Result
Searches	/*string*/	Searches for first occurrence of *string*, beginning at ".+1".
	?*string*?	Searches backwards for first occurrence of *string*, beginning at ".-1".
	g/*string*/	Searches for *every* line containing *string*.
	v/*string*/	Searches for every line **not** containing *string*.
Regular Expressions	.	(period) Matches any single character.
	\	Removes the special significance of the following character (except "(", ")", or a digit).
	[*chars*]	Matches if the character found is in the set of *chars*.
	^	Matches at the beginning of the line.
	$	Matches at the end of the line.
	*	Matches a sequence of 0 or more occurrences of the preceding character.
	&	Used on the right side of a substitution command, "&" matches the string on the left side.
	\(\)	Backslash-parentheses surround an identifiable part of the search string (left side) in a substitution. On right side of the substitution, \1 represents item in first pair of backslash parentheses, \2 represents item in second pair, etc.

Section 6
MAINTAINING FILES (MAKE)

ILLUSTRATIONS

TABLES

Section 6

MAINTAINING FILES (MAKE)

INTRODUCTION

When software programs are broken down into small, manageable files, a change to any of those files usually means that other files need to be updated before you can get a clean new version. If you maintain the project by hand, every change requires that you reprocess all the files (which is wasteful and time-consuming), or that you identify only the files that need redoing and do those (which is error-prone and time-consuming).

The TNIX **make** utility program eases the task of updating and maintaining files. **Make** can automatically find out which files are affected by a change, and do whatever is necessary to update those files: compile, assemble, link, install, etc. **Make** can also perform housecleaning tasks: remove temporary files, run test scripts, print listings, change the "last modified" date of files, archive, and so on.

Make needs to be told what files affect other files, and what to do if a file is out of date. You do this just once by creating a "makefile" (also known as a description file). The makefile spells out which files "depend on" other files, which commands update a file, and other information (such as how to generate files according to your suffix conventions).

Once you have created the makefile for a set of files, the simple command **make** updates all affected files. You never again have to manually compile, assemble, or link. (You can also run a **make** command in the background.)

This section presents the following topics:

- **The Make Process**. Shows the steps involved in using **make**, an example of a set of files, and a makefile that simplifies maintenance of the files.
- **The Makefile**. Explains the parts of a makefile.
- **Invoking Make**. Explains how to invoke the TNIX **make** program.
- **Applications**. Shows examples of using **make** to maintain listings and archives.
- **Reference Summary**. Contains tables summarizing **make** features and special characters.

THE MAKE PROCESS

The **make** process includes the following steps:

1. Collect the files that depend on each other, or that are interrelated, into a single directory. (By default, **make** looks in the current directory for any files it needs and places in it any files it creates.)
2. Using a text editor, create a makefile in that same directory. You can name the makefile "makefile" or a name of your own choosing (**make** automatically looks for a file named "makefile").
3. Whenever you need a clean version of any of the files, such as after an editing session, issue the single command

   ```
   $ make
   ```

4. (Optional) You may reedit the makefile if you change the names of your files, or if you change the commands needed to update the files.

An Example

Suppose you have a program called *ezsoftware*. For this program to be up-to-date, six other files must also be up-to-date:

three assembler source files—*main.asm*, *sub1.asm*, and *sub2.asm*
assembled object files—*main.obj*, *sub1.obj*, and *sub2.obj*

Figure 6-1 shows how these files are related.

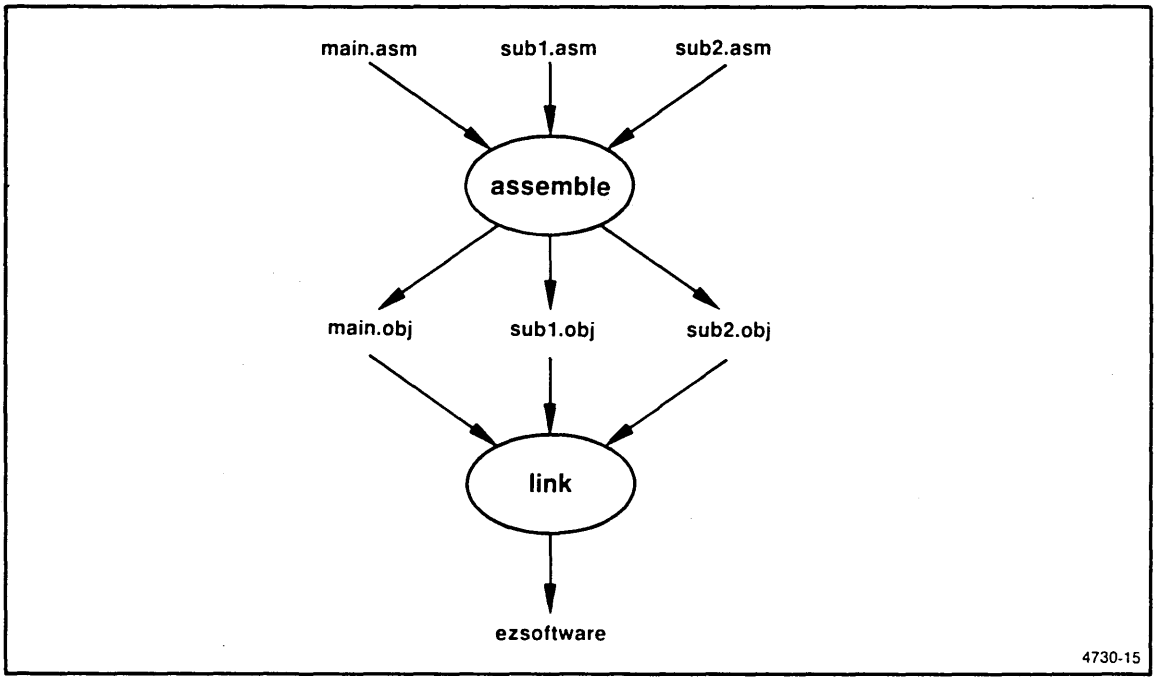


Fig. 6-1. Relationships between files in a software program.

In this example, program **ezsoftware** depends on the current versions of files *main.obj*, *sub1.obj*, and *sub2.obj*. Those three files depend on the current versions of files *main.asm*, *sub1.asm*, and *sub2.asm*. The **make** program provides automatic, programmable updating of all the files.

Figure 6-2 shows a makefile that defines which files in the example in Figure 6-1 depend on which other files, and provides information necessary to update these files.

```
# makefile for example software project
ezsoftware : main.obj subl.obj sub2.obj
        link -o ezsoftware -O main.obj subl.obj sub2.obj
        # command to update ezsoftware if object files
        # have been changed
main.obj : main.asm
        asm main.obj '' main.asm        # command to update main.obj
                                        # if main.asm has been changed
subl.obj : subl.asm
        asm subl.obj '' subl.asm        # command to update subl.obj
                                        # if subl.asm has been changed
sub2.obj : sub2.asm
        asm sub2.obj '' sub2.asm        # command to update sub2.obj
                                        # if sub2.asm has been changed
```

Fig. 6-2. Makefile for example software program.

This makefile defines which files affect other files, and tells **make** how to update the files. For example, the program *ezsoftware* depends on *main.obj, sub1.obj,* and *sub2.obj.* To make program *ezsoftware* completely current, the three object files are linked together if any of them is more recent than the program.

The makefile for the example software project specifies four "targets", or objects to be "made": the program *ezsoftware* and the object files *main.obj*, *sub1.obj*, and *sub2.obj*. Each of the four targets has its own "entry" in the makefile, with each entry containing the name of the target, the list of files the target depends on, and the commands needed to update that target.

In this example, if only the file *main.asm* has been changed, then the **make** command produces the following:

```
$ make
asm main.obj ' ' main.asm
link -o ezsoftware -O main.obj subl.obj sub2.obj
```

Terminology

The following paragraphs describe terms used throughout this section.

Target. The file to be updated (e.g., a new program) or the action to be taken (e.g., installing or printing). The main function of **make** is updating various *targets.*

Makefile (or Description File). A text file that tells **make** which files are out of date with respect to each other, and what to do to make the files current.

Dependency. A file that a target depends on.

Dependency Line. The line in a makefile that associates a target(s) with its dependency(ies). Targets are to the left of the colon(s), dependencies to the right.

Dependency List. The list of files on which a target depends.

Entry. A fragment of a makefile that contains the target, its dependencies, and the commands to update the target.

Suffix Rules. A special type of entry in a makefile that tells **make** the general method to transform files of one suffix to files of another suffix. (For example, to produce an *.obj* file from an *.asm* file, use the **asm** command).

THE MAKEFILE

A makefile contains four kinds of text: entries, which are required, and three kinds of optional information: comments, macros, and suffix rules. The following pages discuss each of these parts of a makefile.

Figure 6-3 shows the contents of a makefile that includes each possible part.

```
# macro definitions
SOURCES = main.asm subl.asm sub2.asm
OBJECTS = main.obj subl.obj sub2.obj
# suffix rule
.SUFFIXES : .obj .asm
.asm.obj :
        asm $*.obj ' ' $<
# entries (two entries in this makefile)
program : $(OBJECTS)
        link -o ezsoftware -O $(OBJECTS)
modules : $(SOURCES)
```

Fig. 6-3. The parts of a makefile.

Makefiles must contain entries, and may also contain macros, comments, and suffix rules. The optional parts can increase the makefile's clarity and flexibility.

Entries

The makefile must have an entry in order for **make** to work. An entry consists of a dependency line and zero or more command lines. In Fig. 6-3, the following fragment is an entry:

```
program : $(OBJECTS)                    # dependency line
        link -o ezsoftware -O $(OBJECTS)    # command line
```

The dependency line contains one or more targets, one or two colons, and zero or more dependencies. Both targets and dependencies are strings of letters, digits, periods, and slashes.

The command line immediately follows a dependency line and begins with a tab. More than one command line may follow a dependency line, but each command line must begin with a tab. The format for an entire entry looks like this:

```
target1 target2 : dep1 dep2 dep3
< tab > command1
< tab > command2
target3 : dep4 dep5
< tab > commands
...
```

When any line in an entry becomes too long, you can continue it by using a backslash: if the last character of a line is a backslash, then **make** interprets the backslash, newline, and following blanks and tabs as a single blank separating successive words.

Use of Colons in Dependency Lines

A single colon requires that the target appear in only one entry. A double colon shows that the target appears in more than one entry. A target must not appear on both a single-colon and a double-colon line.

For example, the following entries are **incorrect**, because *target2* occurs in two different entries:

```
target1 target2 : dep1 dep2
      command1
target2 : dep3
      command2
```

One way to fix this is by consolidating the dependency and command lines with a single colon:

```
target1 target2 : dep1 dep2 dep3
      command1
      command2
```

The disadvantage of this approach is that **make** executes the commands if *target1* is out of date with respect to *dep3*, even though *target1* does not really depend on *dep3*.

A better way to fix this is to use the double colon syntax:

```
target1 target2 :: dep1 dep2
      command1

target2 :: dep3
      command2
```

This way, if *dep3* was modified more recently than *target2*, then **make** will execute *command2* but not *command1*. For an example of double-colon use, see "Applications", later in this section.

Targets That Are Not Files

A makefile target need not be a file at all. Sometimes you may want to construct a makefile so that you can enter commands such as:

```
$ make install
```

Make automatically carries out any commands following the name of a target if there is no file of that name. For example, the following makefile entry copies *resultfile* into a command directory, */bin*:

```
install : resultfile
        cp resultfile /bin
```

If there is no actual file named *install*, every **make install** command copies the file anyway, after creating an up-to-date version of *resultfile*. (The commands for updating *resultfile* must also be present in the makefile.)

Commands

Make can execute any TNIX command. However, you must access shell variables from within a makefile with two dollar signs ($$) instead of the usual one (see "Macros", later in this section).

Commands need not occupy separate lines. You may combine commands into a single line (commands separated by semicolons), or append a command to a dependency line, following a semicolon:

```
target1 : dep1 dep2; command1
```

Details of TNIX Command Execution under the Control of Make. Command lines are executed one at a time, each in its own shell.

In **make**, some "special" shell commands—such as **cd**—don't work across newlines. Assume you have this entry in your makefile:

```
target1 : dep1 dep2
        cd newdir
        ls
```

The **ls** command does not list the contents of *newdir*, but instead lists the directory you were in before the **cd** command.

Make executes each command line within an entirely new shell, unrelated to previous shells. Because the effect of special shell commands (like **cd**) is restricted to the invoking instance of the shell, a special shell command on one line (one side of a newline) does not affect other command lines (the other side of a newline).

To get around this newline problem, use a semicolon. In the following example, the **ls** command does list the contents of *newdir*, because the **ls** command is on the same line as the **cd** command:

```
targetl : depl dep2
      cd newdir; ls
```

Echoing of TNIX Commands. Make displays each command line on the terminal as the command executes. To suppress this echoing throughout a makefile, place the pseudo-target, ".SILENT:", somewhere in the makefile. To suppress echoing for a single command, make the first character of the command "@".

Error Handling. A TNIX command executing abnormally (returning a non-zero exit status) causes **make** itself to terminate. To suppress this automatic termination throughout the makefile, put the pseudo-target ".IGNORE:" somewhere in the makefile. To suppress the automatic termination for a single command, start the command line with a hyphen (following the usual tab).

Similarly, the interrupt and quit signals coming from a TNIX command cause the current target to be deleted. This ensures a clean state for **make** when you run it again. You can protect the target from deletion by making it depend on the pseudo-file, ".PRECIOUS", like this:

```
target : depl dep2 .PRECIOUS
```

NOTE

Your assembler may return non-zero exit status, causing ***make*** *to quit—but only after the assembler has already created a defective output file with the current date on it. This leaves the appearance that the assembler's output file was correctly updated. If your assembler sends error messages to the terminal during execution of* ***make,*** *remove the output files:*

```
$ rm objfile listfile
```

Comments

The pound sign starts a comment: characters from a pound sign (#) to the end of a line in a makefile are ignored. **Make** also ignores blank lines in makefiles. In Fig. 6-3, the line "# macro definitions" is a comment.

Macros

You can use macros to simplify your makefile. For example, if your makefile has several different entries in a makefile that depend on the same set of files, you can define a macro to equal that set of files and refer to the macro name when making the entries. That way, if you have to change anything in the set of files, you need to edit only the single macro definition line instead of all the entries that use the set of files.

A macro is a string variable, and has the form "string1 = string2". In Fig. 6-3, the following line is a macro definition:

```
SOURCES = main.asm subl.asm sub2.asm
```

The following are valid macro definitions:

```
ASSEMBLER=/bin/asm
abc = filel file2
LIBES =
```

(The last definition assigns the null string to "LIBES", so that you can later pass a value to that macro when invoking **make**. For an example, see "Assigning Values to Macros When Invoking Make", later in this section.) **Make** does not expand the filename pattern-matching characters, "*", "?", and "[...]" in macro definitions.

To invoke a macro, precede its name with a dollar sign. A macro name longer than one character must be placed within parentheses (or braces) when invoked.

The following fragment of a makefile defines and invokes two macros, OBJECTS and LIBES. When a **make** command is issued, the three object files are linked with the library file:

```
OBJECTS = x.obj y.obj z.obj
LIBES = lib.misc
prog : $(OBJECTS)
      link -o prog -0 $(OBJECTS) $(LIBES)
```

Literal Dollar Sign. To specify a literal dollar sign in a makefile, use $$. For example, if you need to access the value of a shell variable from within a makefile, use "$$" in place of the single dollar sign that normally precedes a shell variable name. (The makefile command line "echo $$HOME" echoes the name of your login directory.)

Assigning Values to Macros when Invoking Make. You may also define macros on the **make** invocation line:

```
$ make "name = string2"
```

Such macros override definitions in the makefile. For example, assume a makefile contains the following entry:

```
TESTER = stdtest
test : x
      $(TESTER) x
```

The **make** command invocation

```
$ make "TESTER = newtest"
```

executes the command

```
newtest x
```

Internal Macros. Make defines macros of its own, whose values may change as **make** executes. Two of these macros are useful in writing makefile entries:

$? The list of file names found to be more recently modified than the target currently being "made". This string is a subset of the names on which the target depends.

$@ The name of the file being "made"—the current target.

The following makefile entry illustrates these internal macros:

```
program : $(OBJECTS)
        link -O $(OBJECTS) -o $@
file1.po : $(SOURCES)
        pr $?
```

Make expands "$@" in this example into "program"—the target of the **link** command. The command line "pr $?" prints the outdated source files from the dependency list for target *file1.po*.

The other macros internal to **make** are relevant in the definition of suffix rules. They are summarized in Table 6-1 at the end of this section.

Suffix Rules

Suffix Rules. Suffix rules tell **make** the general method to transform files of one suffix to files of another suffix. Two of **make**'s internal macros are used in setting up suffix rules:

$< The name of the file used to invoke a suffix rule.

$* The prefix shared by the target and the files on which it depends.

In Fig. 6-3, the following lines are a "suffix rule" that tell **make** how to generate an *.obj* file from an *.asm* file:

```
.SUFFIXES : .obj .asm
.asm.obj :
        asm $*.obj ' ' $<
```

Make has a set of default suffix rules, which are described at the end of this section. To avoid suffix rule conflicts, add the following line to your makefile:

```
.SUFFIXES :
```

This clears any previously defined suffixes. To create new suffix rules, simply add a second ".SUFFIXES :" entry containing the new suffixes.

A blank must appear between each suffix in the suffix list, but must not appear between suffix-pairs or as a target on a dependency line.

The order of the suffixes is significant.

- When listing the suffixes after the ".SUFFIXES :" entry, list the output suffix first, and source suffix last.
- When specifying the pair of suffixes as the target in a dependency line, list the suffix of the source first, and the suffix of the output second.

Sample Suffix Rules. If you have an 8560/8086 Pascal compiler and an 8500 Modular MDL Series assembler, then the following makefile fragment is an example of a valid suffix rule:

```
.SUFFIXES :     # delete existing rules
.SUFFIXES : .po .ps .obj .asm

PASFLAGS=       # put in your compiler flags
ASMFLAG=        # put in your assembler flags

.ps.po :
    pas $(PASFLAGS) $<
.asm.obj :
    asm $*.obj $(ASMFLAG) $<
```

Note that the PASFLAGS and ASMFLAG macros are left null so that you can assign non-null values to these macros when invoking **make**. For example,

```
$ make "PASFLAGS= -l -s" prog
```

causes the Pascal compiler to send a source listing to standard output, and suppresses the compiler optimizer. If you use different flags at different times, it is convenient to put only the most common requirements into suffix rules.

INVOKING MAKE

The following paragraphs describe the TNIX **make** command.

SYNTAX

make [-**iknrst**] [*"macro=string"*] [**–f** *file*]... [*target*] ...

PARAMETERS

none	When **make** is entered with no parameters, it searches for a file named *Makefile* or *makefile* in the current directory, and executes commands according to that file.
-i	(Ignore) Equivalent to the special entry ".IGNORE:". Ignores errors from TNIX commands in the makefile.
-k	(Kill work on current entry). When a command has not completed as expected, abandons work on the current entry, but continues on branches that do not depend on the current entry.
-n	(No work) Traces and prints, but does not execute the commands needed to update the targets.
-r	(Clear suffixes) Equivalent to an initial special entry ".SUFFIXES:" with no list. Clears any previously defined suffixes.
-s	(Silent) Equivalent to the special entry ".SILENT:". Suppresses echoing of TNIX commands in the makefile.
-t	(Touch) Updates the modified date of targets, without executing any commands.
macro=string	Insert *string* wherever *macro* is invoked within the specified makefile. Overrides definition of *macro* within the makefile.
-f file	Executes commands from the specified file rather than the default name *makefile* or *Makefile*. More than one **–f** option may appear.
target	The name of the file (target) to be updated or modified, or the action to be taken. If no target is given, **make** uses the first target in the current makefile.

EXPLANATION

Make executes commands in *makefile* to update one or more target *files.*

Make updates or creates a target if

1. that target depends on files that have been modified since the target was last modified, or
2. if the target does not exist (for example, if the target is not specifically a file).

EXAMPLES

```
$ make -n ezsoftware
```

Prints out, but does not execute (**-n**), the commands needed to produce an up-to-date version of the target *ezsoftware*. The default makefile name *makefile* is assumed.

```
$ nohup make -s &
```

Runs **make** in the background, working on the first target in the default makefile named *makefile* or *Makefile*, and suppressing (**-s**) the echoing of any commands executed from the makefile. The command **nohup** makes sure that **make** continues executing even if you log out.

```
$ make program -f newmakefile
```

Runs **make** to update file *program*, using file named *newmakefile* as the makefile.

APPLICATIONS

Make can provide many services besides assembling, compiling, or linking files. The following examples show how **make** can maintain printed listings and archives.

Maintaining Printed Listings. The following makefile entry maintains up-to-date printed listings:

```
print : file1 file2...filen
        pr $?           # print only those files that have been changed
                        # since the last update of the file print.
        touch print     # The TNIX touch command creates a file,
                        # or updates its "last modified" time.
```

In this example *print* is simply a file name. But the TNIX file system keeps track of it along with all other files, so that its "last modified" time can be used as a marker against which to determine the change status of other files.

Maintaining Archives. The TNIX **libgen** command consolidates groups of files into single archive or library files. For example,

```
libgen -h lib -r file1 -r file2
```

replaces *file1* and *file2* in the *lib* library with new versions found in *file1* and *file2*. By using **make** and **libgen** together, you can easily update files and incorporate them into an archive.

Assume, for example, that you have a *lib* file and a number of Pascal source files, *file1.ps, file2.ps*, etc. If you are willing to keep all the object files in the file system after they are copied into *lib*, then the following fragment will do:

```
lib : filel.po file2.po...
      libgen -h lib -r $?
filel.po : filel.ps
      pas filel.ps
file2.po : file2.ps
      pas file2.ps
...
```

Make compiles each source file that is outdated, and copies the new version into *lib*. There are now two copies of each object file: one in *lib* and one in a *.po* file. If you do not wish to keep the extra object files, you can use the double-colon syntax (see "The Makefile", earlier in this section):

```
lib :: filel.po
      pas filel.ps
      libgen -h lib -r filel.po
      rm filel.obj
lib :: file2.po
      pas file2.ps
      libgen -h lib -r file2.po
      rm file2.po
```

Once a source file is compiled and the object file copied into *lib*, the object file is removed from the file system.

REFERENCE SUMMARY

Table 6-1 shows the characters that have special meaning to **make**.

Table 6-1
Make Special Characters

Special Character	Function
#	Starts a comment. Anything on the rest of the line in a makefile is ignored by **make**.
:	Separates the parts of a dependency line: names to the left of the colon are *targets,* names to the right of the colon are the files on which the target depends, or *dependencies.*
::	One of the targets (names to the left of the double colon in a dependency line) appears in more than that one dependency line.
$@	The name of the file being "made"—the current target.
$*	Used in suffix definitions. When **make** invokes a suffix rule, "$*" stands for the basename shared by the file used to invoke the suffix rule and the file to be created by the suffix rule.
$<	Used in suffix definitions. Refers to the name of the file used to invoke the suffix rule.
$?	The list of dependency files that have been changed more recently than the target (the file currently being "made").
-	Placed before a command in an entry. Causes **make** to ignore the exit status of that command. "-" and "@" can appear together, in either order.
@	Placed before a command in an entry. Causes that command to run without echoing. "-" and "@" can appear together, in either order.

Table 6-2 shows the reserved words (pseudo-targets) that have special meaning to **make**.

Table 6-2
Make Reserved Words

Reserved Word	Function
.DEFAULT	If a file must be "made" but there are no explicit commands or relevant suffix rules given, then **make** will execute the commands associated with the pseudo-target ".DEFAULT:".
.IGNORE	Errors returned by TNIX commands are ignored.
.PRECIOUS	In case of error, the current target is not deleted from the file system. This entry follows the colon (or double-colon) on a dependency line.
.SILENT	Command lines are not echoed before execution. Applies to all commands in the makefile.
.SUFFIXES	Adds the following suffixes to the list of **make**'s default suffix rules. This entry without any following suffixes clears the current suffix list.

Default Suffix Rules

The default suffix rules apply to the software development tools in the optional 8560 Series MUSDU Native Programming Package. You need to be aware of these rules, even if your system does not have this package, so that you can avoid conflicts between your own filename suffixes and the default suffix rules.

Following are the default suffix rules for **make**:

```
.SUFFIXES : .out .o .c .f .e .r .y .yr .ye .l .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=f77
RFLAGS=
EC=f77
EFLAGS=
FFLAGS=
```

```
LOADLIBES=
.c.o :
	$(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
	$(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
	$(AS) -o $@ $<
.y.o :
	$(YACC) $(YFLAGS) $<
	$(cc) $(CFLAGS) -c y.tab.c
	rm y.tab.c
	mv y.tab.o $@
.yr.o :
	$(YACCR) $(YFLAGS) $<
	$(RC) $(RFLAGS) -c y.tab.r
	rm y.tab.r
	mv y.tab.o $@
.ye.o :
	$(YACCE) $(YFLAGS) $<
	$(EC) $(RFLAGS) -c y.tab.e
	rm y.tab.e
	mv y.tab.o $@
.l.o :
	$(LEX) $(LFLAGS) $<
	$(CC) $(CFLAGS) -c lex.yy.c
	rm lex.yy.c
	mv lex.yy.o $@
.y.c :
	$(YACC) $(YFLAGS) $<
	mv y.tab.c $@
.l.c :
	$(LEX) $<
	mv lex.yy.c $@
.yr.r :
	$(YACCR) $(YFLAGS) $<
	mv y.tab.r $@
.ye.e :
	$(YACCE) $(YFLAGS) $<
	mv y.tab.e $@
.s.out .c.out .o.out :
	$(CC) $(CFLAGS) $< $(LOADLIBES) -o $@
.f.out .r.out .e.out :
	$(EC) $(EFLAGS) $(RFLAGS) $(FFLAGS) $< $(LOADLIBES) -o $@
	-rm $*.o
.y.out :
	$(YACC) $(YFLAGS) $<
	$(CC) $(CFLAGS) y.tab.c $(LOADLIBES) -ly -o $@
	rm y.tab.c
.l.out :
$(LEX) $<
$(CC) $(CFLAGS) lex.yy.c $(LOADLIBES) -ll -o $@
	rm lex.yy.c
```

Section 7
COMMUNICATION WITH 8540S AND 8550S

ILLUSTRATIONS

Section 7

COMMUNICATION WITH 8540S AND 8550S

INTRODUCTION

This section explains how to use your 8560 with a TEKTRONIX 8540 Integration Unit or 8550 Microcomputer Development Lab. (Throughout this section, the term "workstation" refers to an 8540 or 8550.)

This section includes the following topics:

- **TERM Mode**. Explains TERM mode, the mode of communication between an 8560 and a workstation. Also explains the COM interface.
- **System Configurations**. Explains the ways in which your 8560, terminal, and workstation may be interconnected, and how commands and other information are passed back and forth in each configuration.
- **Establishing Communication**. Summarizes the steps in establishing and terminating TERM mode communication between the 8560 and the workstation in each configuration.
- **Special Considerations**. Explains how TERM mode affects certain workstation operations.
- **Transferring Files and Programs**. Provides procedures for transferring files and object code between the 8560 and the workstation.

NOTE

This section does not cover certain hardware and software configuration procedures that must be performed by the system manager before TERM mode communication can take place. These procedures are described in the ***8560 Series System Manager's Guide.***

TERM MODE

TERM mode is the recommended method of communication between the 8560 and a workstation. TERM mode enables you to use the resources of your 8560 and your workstation simultaneously.

In TERM mode, you enter workstation commands (to perform software-hardware integration and hardware debugging) as if they were 8560 commands. The 8560 recognizes the workstation commands and passes them on to the workstation for processing. The workstation performs the requested function and responds to the 8560. The results of the command are then available for further processing.

For example, the workstation command

```
$ d 0 0ff
```

dumps the contents of program memory locations 0-0FF to your terminal. In TERM mode, you may enter this command as if you were communicating only with the workstation. You can also use the TNIX shell's ability to redirect the input and output of commands, and perform complex command files ("shell programs"). For example, the command

```
$ d 0 0ff ¦ lplr
```

dumps the contents of locations 0-0FF to the lineprinter, while

```
$ d 0 0ff >dumpfile
```

writes the contents of locations 0-0FF into the file *dumpfile*.

COM Interface

In addition to TERM mode, the 8540 and 8550 can also communicate with an 8560 through the **COM** command. (COM Interface software is standard with the 8550, optional with the 8540.)

By means of the TNIX commands **mload** and **uload**, the 8560 can act as host to an 8540 or 8550 running the **COM** command. However, since TERM mode is the preferred mode of communication between an 8540 or 8550 and an 8560, COM is not discussed in this section. For more information about using COM, refer to the online manual pages for **mload** and **uload**, and to the *System Users Manual* for your workstation. The *8560 Series System Manager's Guide* gives instructions for using COM to establish 8560-workstation communications via modems.

SYSTEM CONFIGURATIONS

In TERM mode, your 8560 operates in one or more of the following configurations:

A. A terminal connected to an 8540, which is in turn connected to the 8560.

B. A terminal connected to an 8550, which is in turn connected to the 8560.

C. A terminal connected to the 8560, which is in turn connected to an 8540.

D. A terminal connected to the 8560, which is in turn connected to an 8550.

Each configuration is referred to by the letter A, B, C, or D throughout the rest of the section.

In configurations A and B, the workstation is physically connected between the 8560 and the terminal, but is transparent to you. It is as though your terminal was connected directly to the 8560. Figure 7-1 shows how information is passed in configurations A and B.

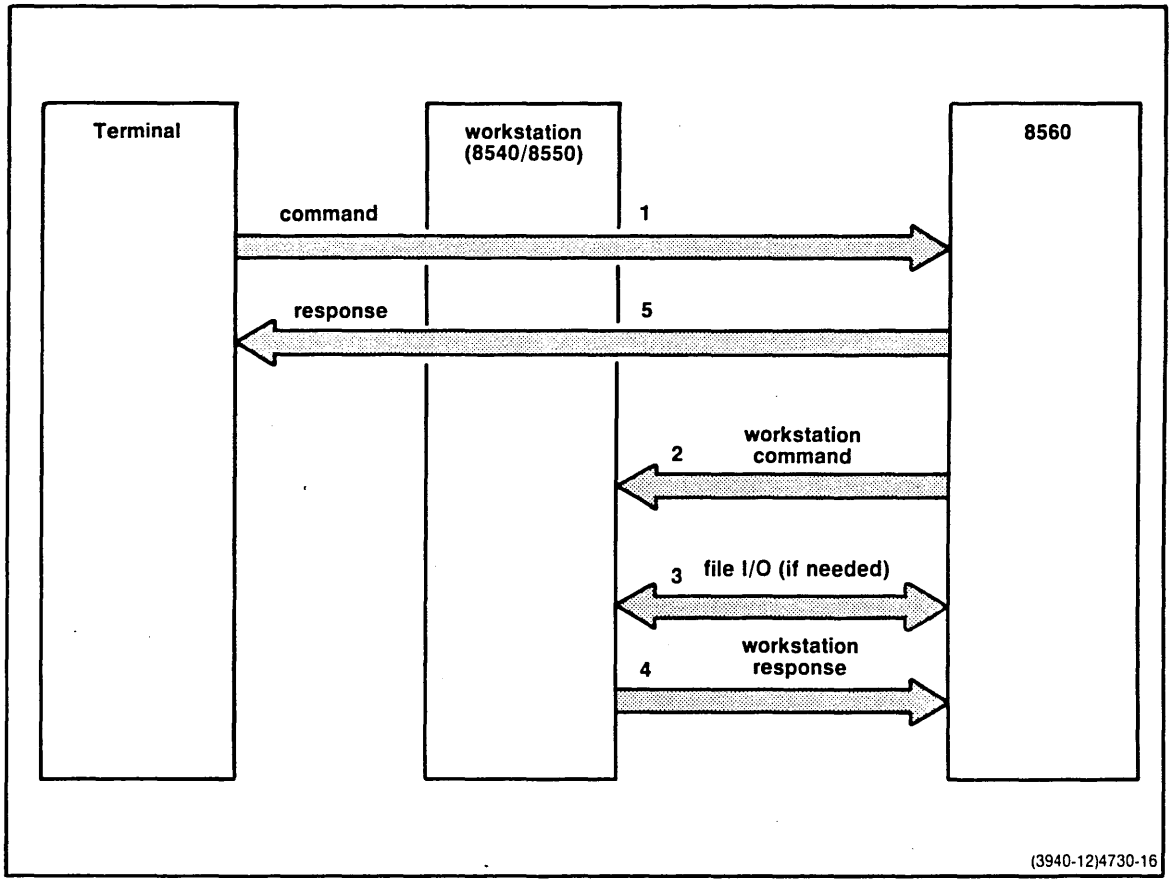


Fig. 7-1. TERM mode operation in configurations A and B.

In configurations A and B, when you enter a workstation command at your terminal, the following steps occur:

1. The 8560 reads the command and recognizes it as a workstation command.
2. The command is sent to the workstation, where it is actually executed.
3. Any file I/O is sent between the 8560 and the workstation.
4. The command output is sent back to the 8560 ...
5. ... and then to the terminal.

In configurations C and D, the terminal and workstation are each connected separately to the 8560. Here, you must specify the workstation on which you want the workstation commands executed before you start sending commands to it. Figure 7-2 shows how information is passed in configurations C and D.

In configurations C and D, you can control more than one workstation from the same terminal by specifying which of the eight HSI I/O ports a given workstation is attached to. For example, to begin directing commands to an 8540 connected to HSI I/O port 4, you would enter the following line:

```
$ IU=4; export IU
```

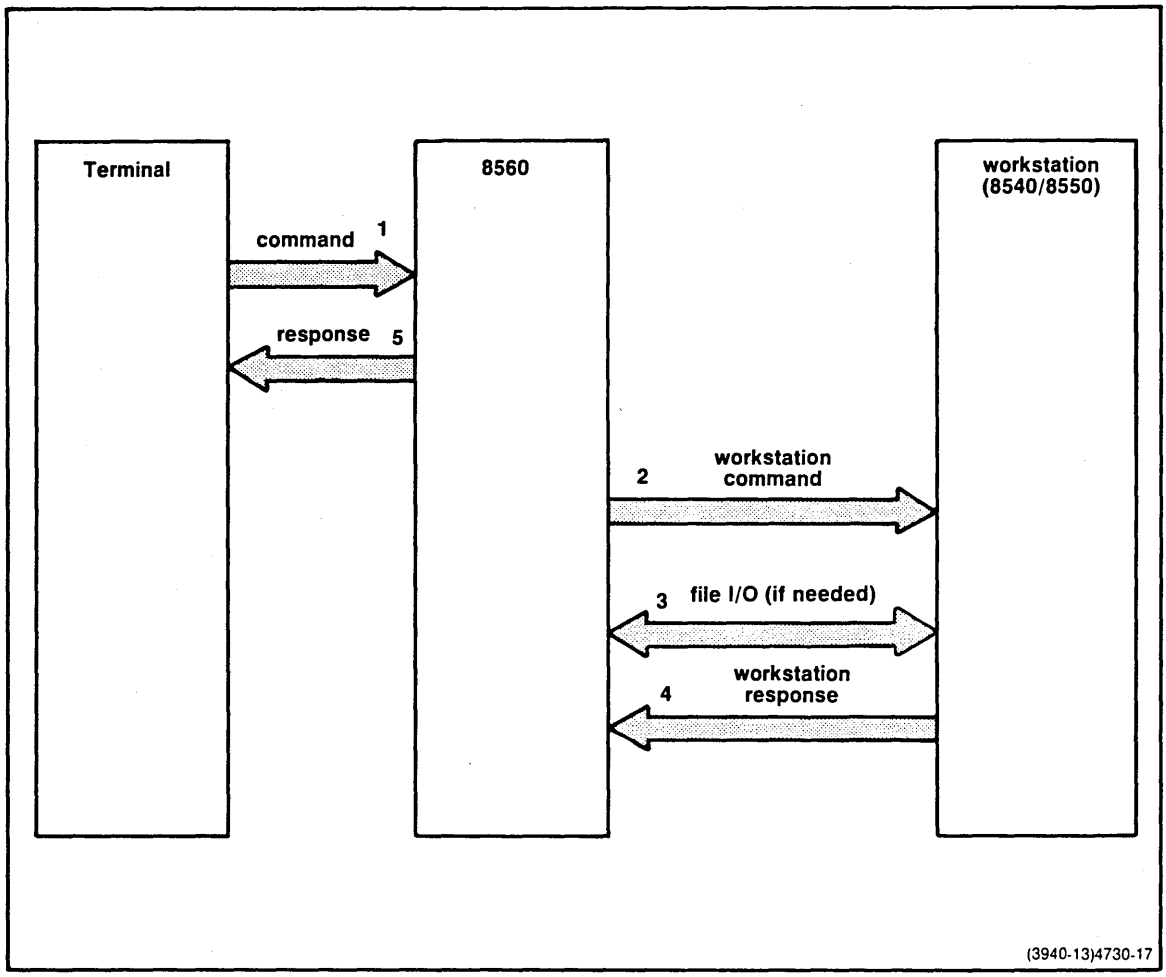


Fig. 7-2. TERM mode operation in configurations C and D.

In configurations C and D, when you enter a workstation command at your terminal, the following steps occur:

1. The 8560 reads the command and recognizes it as a workstation command.
2. The command is sent to the workstation, where it is actually executed.
3. Any file I/O is sent between the 8560 and the workstation.
4. The command output is sent back to the 8560 ...
5. ... and then to the terminal.

ESTABLISHING COMMUNICATION

This subsection shows how to establish TERM mode communication between the 8560 and the workstation in each of the four previously defined configurations.

Configuration A: Terminal-8540-8560

The following tasks must be performed before you can use TERM mode in configuration A:

1. The terminal, 8560, and 8540 must be properly interconnected, as shown in Fig. 7-3. Notice that the cable from the 8560 connects to the "HSI" port on the 8540. This task should be performed by the system manager during the system configuration process, as described in the *8560 Series System Managers Guide*.
2. The 8560 HSI I/O port that connects to the 8540 must be properly configured for TERM mode communication. This task should be performed by the system manager during the system configuration process, as described in the *8560 Series System Managers Guide*.
3. Start up the 8540 and enter the command **config term**. You may then log in to TNIX as you would through any other terminal. (You may have to press the RETURN key or the BREAK key a few times to obtain the TNIX login prompt.)

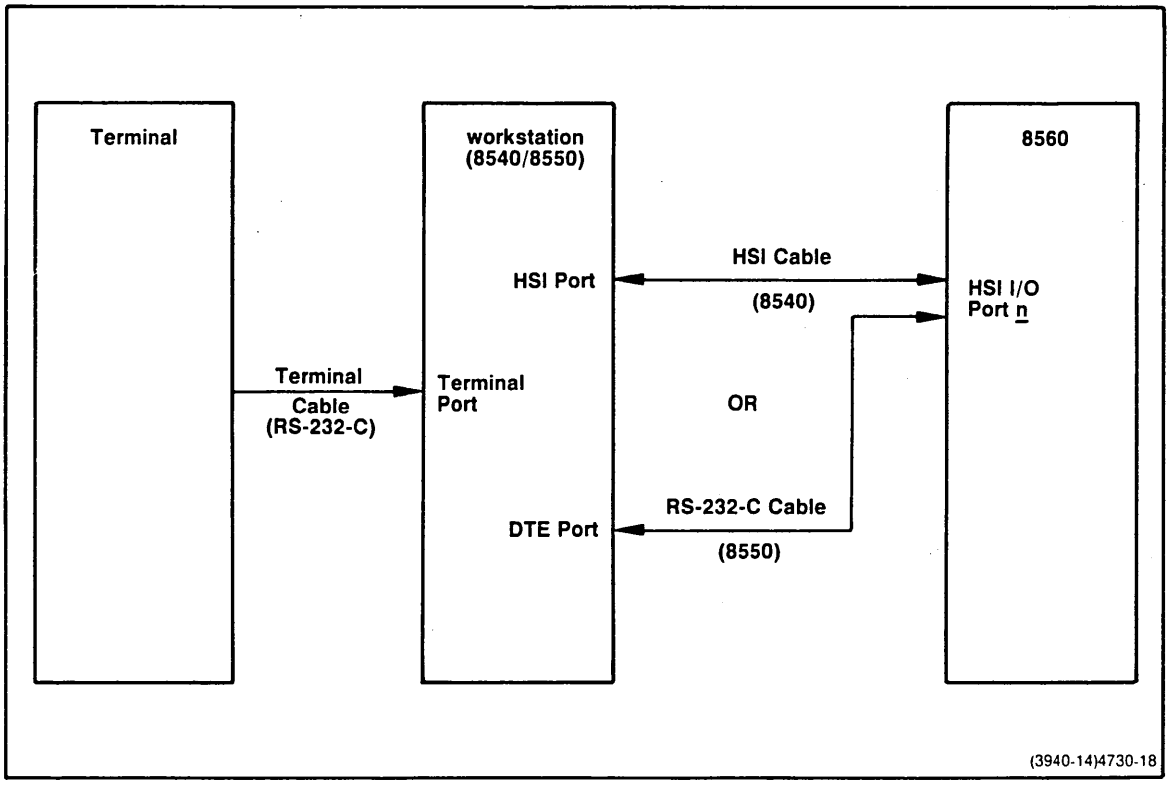


Fig. 7-3. TERM mode interconnection diagram for configurations A and B.

Before you can use TERM mode, the 8560's HSI port must be properly configured by qualified servicing personnel for TERM mode communication with the 8540 or 8550. Instructions are provided in the *8560 Series System Manager's Guide*.

Configuration B: Terminal-8550-8560

The following tasks must be performed before you can use TERM mode in configuration B:

1. The terminal, 8560, and 8550 must be properly interconnected, as shown in Fig. 7-3. Notice that the cable from the 8560 connects to the "DTE" port on the 8550. This task should be performed by the system manager during the system configuration process, as described in the *8560 Series System Managers Guide.*
2. The 8560 HSI I/O port that connects to the 8550 must be properly configured for TERM mode communication. This task should be performed by the system manager during the system configuration process, as described in the *8560 Series System Managers Guide.*
3. Start up the 8550 and enter the command **config term t=7**. You may then log in to TNIX as you would through any other terminal. (You may have to press the RETURN key or the BREAK key a few times to obtain the TNIX login prompt.)

Configuration C: Terminal-8560-8540

The following tasks must be performed before you can use TERM mode in configuration C:

1. The terminal, 8560, and workstation must be properly interconnected, as shown in Fig. 7-4. Notice that the cable from the 8560 connects to the "HSI" port on the 8540. This task should be performed by the system manager during the system configuration process, as described in the *8560 Series System Managers Guide.*
2. The 8560 HSI I/O port that connects to the 8540 must be properly configured for TERM mode communication. This task should be performed by the system manager during the system configuration process, as described in the *8560 Series System Managers Guide.*
3. Log in to TNIX at the terminal and enter the following line (in place of **n**, type the number of the HSI I/O port that the 8540 is connected to):

   ```
   $ IU=n; export IU
   ```

   Before the 8540 can accept commands from the 8560, the 8540 must be in TERM mode. If there is no terminal attached to the 8540, you will want to configure the 8540 to enter TERM mode automatically every time you start it up. To do so, temporarily attach a terminal to the 8540 and enter the following commands, which place the **config term** command in the 8540's STARTUP string:

   ```
   > STARTUP='config term'
   > permstr -d STARTUP
   > permstr STARTUP
   ```

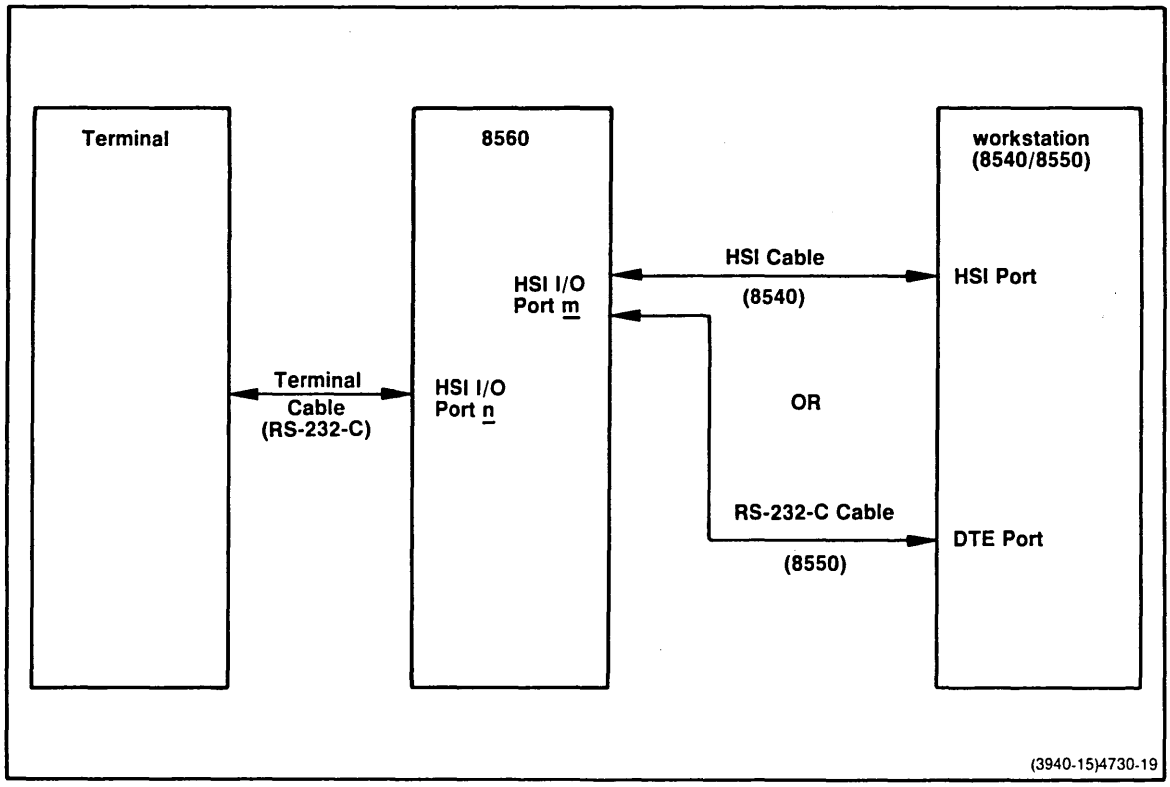


Fig. 7-4. TERM mode interconnection diagram for configurations C and D.

Before you can use TERM mode, the 8560's HSI port must be properly configured by qualified servicing personnel for TERM mode communication with the 8540 or 8550. Instructions are provided in the the *8560 Series System Manager's Guide.*

Configuration D: Terminal-8560-8550

The following tasks must be performed before you can use TERM mode in configuration D:

1. The terminal, 8560, and 8550 must be properly interconnected, as shown in Fig. 7-4. Notice that the cable from the 8560 connects to the "DTE" port on the 8550. This task should be performed by the system manager during the system configuration process, as described in the *8560 Series System Managers Guide.*
2. The 8560 HSI I/O port that connects to the 8550 must be properly configured for TERM mode communication. This task should be performed by the system manager during the system configuration process, as described in the *8560 Series System Managers Guide.*

3. Log in to TNIX at the terminal and enter the following lines (in place of **n**, type the number of the HSI I/O port that the 8550 is connected to):

```
$ stty IU >/dev/ttyn
$ IU=n; export IU
```

Before the 8550 can accept commands from the 8560, the 8550 must be placed in TERM mode with the command **config term t=7**. If there is no terminal attached to the 8550, you will want to configure your DOS/50 system disk to place the 8550 in TERM mode automatically every time you start it up. To do so, temporarily attach a terminal to the 8550 and create a file called STARTUP that contains the following line:

```
config term t=7
```

Make sure that this file resides in the root directory of the DOS/50 system disk.

Terminating Communication

Logging out from TNIX does **not** take the workstation out of TERM mode. In configurations C and D (in which the terminal is attached to the 8560), you may want to leave the workstation in TERM mode so that someone at another terminal can use it. In configurations A and B, if you want to log out and return the workstation to LOCAL mode, enter the following command line:

```
$ config local; logout
```

The **config** command tells the workstation to stop passing commands to the 8560, and the **logout** command tells the 8560 to stop accepting commands. Notice that these two commands must be entered exactly as shown if they are to have the desired effect.

Communication Errors

In configurations C and D, the error message "/dev/hsi**n**—I/O error" indicates that the 8560 attempted unsuccessfully to issue a command to a workstation through HSI I/O port **n**. You may receive this message for any of the following reasons:

- The workstation is not properly properly connected to the 8560.
- The HSI I/O port to which the workstation is attached is configured incorrectly.
- The workstation is not in TERM mode, or the **config** command was somehow incompatible with the configuration of the 8560's HSI I/O port.
- The IU shell variable is set incorrectly.
- You inadvertently entered a workstation command and there is no place to send it.

SPECIAL CONSIDERATIONS

This subsection presents miscellaneous information about TERM mode operation.

Precautions

In TERM mode, TNIX processes every line that you enter. As a result, a command that is acceptable to a workstation in LOCAL mode may have to be altered in order to be processed the same way in TERM mode.

Lowercase Command Names

In TERM mode, workstation commands must be entered in lowercase. All other command elements may be entered in uppercase or lowercase, as permitted by the workstation.

Commas as Delimiters

TNIX does not recognize the comma as a delimiter. For example, in the following workstation command, the commas represent a null first parameter.

```
> svc,,100 200
```

If you enter this command in TERM mode, TNIX attempts to execute a command whose name is **svc,,100**. However, if you insert a space after the command name, TNIX will recognize the command name **svc**, and pass the command to the workstation:

```
$ svc ,,100 200
```

In LOCAL mode, both commands are equivalent.

Special Characters

Certain characters that have special meanings to the TNIX shell must be made to appear non-special, so that the shell does not perform unwanted transformations on the command line. For example, if you enter the command

```
$ p 100 -a "some text"
```

the shell will strip out the quote marks, and the following (erroneous) line will be sent to the workstation:

```
p 100 -a some text
```

To avoid this problem, "escape" any special characters with backslashes:

```
$ p 100 -a \"some text\"
```

Refer to the *Shell Programming* section of this manual for a summary of shell punctuation.

Redirection of Standard Input and Output

Like TNIX, OS/40 and DOS/50 allow you to redirect standard I/O by using the symbols ">" and "<". For example, in LOCAL mode, the following command dumps memory locations 0-0FF to the workstation's line printer:

```
> d 0 0ff >LPT
```

In TERM mode, however, the TNIX shell interprets the ">LPT" construct before the command is passed to the workstation. If you enter this command in TERM mode, TNIX sends the command **d 0 0ff** to the workstation and stores the output in an 8560 file called *LPT*. If you want to direct the output to the workstation's printer in TERM mode, you must use a backslash to hide the ">" symbol from the shell:

```
$ d 0 0ff \>LPT
```

To direct the output to the 8560's printer, you enter:

```
$ d 0 0ff : lplr
```

Command Prefixes "8540" and "8550"

When a workstation command has the same name as a TNIX command, you must use "8540" or "8550" to prefix the workstation command name. For example, in TERM mode the **as** command invokes the assembler in the optional Native Programming Package. To use the workstation's **as** (ASsign) command, you must use the "8540" or "8550" prefix:

```
$ 8540 as 1 CON0
```

At the time of this printing, the following other workstation command names conflict with TNIX command names:

format, asm, link, libgen, ehex, and lstr

You can also prefix a command line with the word "8540" or "8550", to send the entire command line to the workstation. For example, the command

```
$ 8550 modelsetup
```

sends the command **mode1setup** to the workstation, presumably invoking an 8550 command file by that name.

NOTE

The "8540" or "8550" command prefix does ***not*** *prevent the command line from being processed by the TNIX shell before it is sent to the workstation.*

Command Files

As noted previously, you can use a command line such as

```
$ 8550 cmdfilename parm1 parm2
```

to execute a command file on the 8550 in TERM mode. Such a file is executed by the 8550 as if the 8550 were in LOCAL mode. The file must contain only DOS/50 commands—the 8550 cannot send TNIX commands back up to the 8560.

On the other hand, a command file ("shell program") on the 8560 may contain a mixture of TNIX commands and workstation commands. For example, the command file

```
for i in 0 1000 2000
do
  d $i $i+0ff
done
```

executes the following workstation commands:

```
d 0 0+0ff
d 1000 1000+0ff
d 2000 2000+0ff
```

If a workstation command fails, the error code is returned as the command's "exit status", which can be used by subsequent commands in the shell program. (Refer to the *Shell Programming* section of this manual for more details. Error codes are explained in the *Error Messages* sections of the 8540 and 8550 *System Users Manuals*.)

Service Calls and I/O Channels

A program that is running on the emulator in the 8540 or the 8550 can access files by means of **service calls** (SVCs). You can also use workstation commands such as **as** and **cl** to open and close I/O channels that are used by these service calls.

On the 8540, any reference to a filename (other than a standard device name such as CONI or LPT) in a service call or OS/40 command is assumed to refer to a file on the 8560. (Such references are valid only when the 8540 is in TERM mode.) For example, the following command assigns channel 5 to the 8560 file *myfile*:

```
$ 8540 as 5 myfile
```

The 8550, on the other hand, has its own file system. Any reference to a filename in an 8550 service call or command must refer to a file on the 8550. The only way the 8550 can access files on the 8560 is through the standard input, standard output, and standard error channels.

Standard I/O Channels

Workstation I/O channels 8, 9, and 10 are assigned to standard input, standard output, and standard error, respectively. In TERM mode, CONI is equivalent to channel 8, and CONO is equivalent to channel 9. For example, if you enter the command

```
$ g >logfile
```

the 8560 file *logfile* captures any text written to channel 9 or to CONO by the program, as well as any output resulting from the **tra** (TRAce) command or other debugging utilities. Output to *logfile* ends when program execution stops for any reason.

TRANSFERRING FILES AND PROGRAMS

The following paragraphs show how to transfer files and object code between an 8560 and a workstation in TERM mode.

Downloading a Program from an 8560 to an 8540 or 8550

The following command copies a program from a file named *tnix.lo* on the 8560 into the workstation's program/prototype memory:

```
$ lo <tnix.lo
```

The *tnix.lo* file must be in TEKTRONIX A or B Series load module format, as produced by a TEKTRONIX assembler or linker, or by the 8540/8550 SAV command.

Uploading a Program from an 8540 or 8550 to an 8560

The following command saves the contents of the workstation's program/prototype memory locations 0-1FF and 1000-2FFF into an 8560 file called *tnix.sav*, in TEKTRONIX B Series load module format. The transfer address in *tnix.sav* is set to 100 (hexadecimal).

```
$ sav -l 0 1ff 1000 2fff 100 >tnix.sav
```

This code can be loaded back into memory using the previous "Downloading" procedure.

Downloading a File from an 8560 to an 8550

The following command copies an 8560 file called *tnix.file* to the 8550 and gives it the name *dos.file*:

```
$ con -b <tnix.file \>dos.file
```

If *tnix.file* is a text file, be sure to omit the **-b** parameter so that each TNIX end-of-line character (linefeed) is converted to a DOS/50 end-of-line character (carriage return).

Uploading a File from an 8550 to an 8560

The following command copies an 8550 file called *dos.file* to the 8560 and gives it the name *tnix.file*.

```
$ con -b dos.file >tnix.file
```

If *dos.file* is a text file, be sure to omit the **-b** parameter so that each DOS/50 end-of-line character (carriage return) is converted to a TNIX end-of-line character (linefeed).

Section 8
KEYSHELL

TABLES

Section 8

KEYSHELL

INTRODUCTION

Keyshell, introduced in the Learning Guide section of this manual, is an interface to TNIX that enables you to enter commands by pressing function keys as well as by typing the commands literally. Although Keyshell is mostly self-explanatory, experienced TNIX programmers may want to learn about some advanced Keyshell features and about interactions between Keyshell and the regular TNIX command interpreter (the shell).

This section discusses the following topics:

- The interaction between Keyshell and shell commands.
- The automatic invocation of Keyshell at login via the **ksh** command.
- Special Keyshell files.
- How to redraw the Keyshell display.
- The use of the Keyshell command history mechanism, with details about command line editing.

KEYSHELL AND SHELL COMMANDS

You can enter most shell commands as usual while using Keyshell. There are, however, a few exceptions.

Multi-Line Shell Commands

You must precede interactive, multi-line shell commands with "sh". That is, you must execute them as subshells. For example:

```
$ sh
$ for i in ed1.c ed2.c ed3.c
> do
> wc -l $i
> done
$ <CTRL-D>
```

The CTRL-D returns you to the parent shell so that you can resume normal use of Keyshell. All shell commands that use the "case", "while", and "for" constructs must be handled in this way.

You need to use this technique only with shell commands you enter interactively; you can always run shell command files (shell programs) as usual. Note, however, that Keyshell reserves certain names for itself. Do not create shell programs with these names:

exit	save
fs	shiftkey
history	unsave
memused	version
newexp	

Terminal Settings

Keyshell requires that TNIX terminal communication settings remain constant. Therefore, do not alter tty (terminal) characteristics while Keyshell is active. For example, do not use the **stty** (set tty) command while you are using Keyshell.

Similarly, when you are communicating with an 8540 or 8550 in TERM mode, using the **config** command to enter LOCAL mode disables Keyshell.

Retyping Commands

Keyshell does not allow you to use CTRL-K to retype successive characters of the previous command. However, it provides several other ways to repeat and edit commands. See "Keyshell Command History" in this section for details.

"Full-Screen" Commands

Some programs need to use the entire terminal screen, including the area normally reserved for Keyshell function key labels. The ACE and LDE editors, for example, both need to use the full screen. To invoke such a program explicitly while you are in Keyshell, precede the program's name with the special Keyshell command **fs** ("full screen").

For example, to invoke ACE, type:

```
$ fs ace
```

The **fs** command allows **ace** to use the full screen, then redraws the function key labels after **ace** exits.

See "Redrawing the Function Key Labels" in this section for more information about using **fs**.

Logging Out

When Keyshell is invoked automatically, as it usually is, the **logout** command exits Keyshell and logs you out from TNIX. However, when you invoke **ksh** explicitly (the next subsection tells you how), you must leave Keyshell before you can log out.

You can always leave Keyshell and return to unassisted communication with the TNIX shell by pressing the *exit* key on the Keyshell top level or by typing the Keyshell command **exit**.

AUTOMATIC KEYSHELL INVOCATION

The program **ksh** provides the Keyshell interface. Ordinarily, **ksh** is invoked automatically when you log in. This happens because the **login** command executes a command file in your login directory named *.profile*, which contains the **ksh** command. You can prevent this automatic invocation of **ksh** by modifying the copy of *.profile* in your home directory. Comments in *.profile* (lines beginning with ":") explain which lines invoke **ksh**.

For instructions about how to invoke **ksh** explicitly, type **man ksh**. The online manual page explains the **ksh** command options that enable you to control the display of key numbers on the screen and other Keyshell characteristics.

SPECIAL KEYSHELL FILES

Keyshell uses a set of files called *scripts* that determine what happens when you press function keys. These scripts and some other special-purpose files are in your home directory. Like *.profile*, they have names beginning with "." and thus are not ordinarily listed by ls. *Do not remove or alter these files.* If you accidentally change or delete one of them, **ksh** will send you error messages saying that a script is missing or erroneous. If this happens, ask your system manager to rebuild your scripts with the **setuser** command.

REDRAWING THE KEYSHELL FUNCTION KEY LABELS

If you accidentally erase the Keyshell on-screen function key labels, you can usually redraw them by pressing shifted function key 5 (labeled *Redraw Screen* on the plastic keyboard overlay).

If the *Redraw Screen* key does not completely restore the display, use the Keyshell **fs** command like this:

```
$ fs
```

The **fs** command always repaints the entire label display.

KEYSHELL COMMAND HISTORY

Keyshell stores two kinds of "command history" information about the commands you enter:

- **Key history.** Keyshell saves command arguments such as filenames that you type in response to Keyshell prompts. They are displayed as function key labels when it might be appropriate to use them again in the same context.
- **A command history list.** You can use the shifted function keys *History Fwd* and *History Back* to retrieve and execute commands that you have typed or issued via function keys.

The following subsection explains how Keyshell lets you edit your command history list.

Editing Your Command History

The shifted function keys *History Fwd* and *History Back*, in addition to scrolling through the commands you have issued, activate a command history editor that enables you to alter the command line currently on display.

While the history editor is active, your screen displays the message "Command Editor", and the unshifted function keys take on editing functions with new on-screen labels. Table 8-1 describes the keys and their editing functions.

Table 8-1
Command Editing Function Keys

Key	Label	Function
1	ins. char	Inserts a space at the cursor.
2	del. char	Deletes the character at the cursor.
3	ins. mode	Inserts subsequent characters at the current cursor position. Press any function key to leave insert mode.
4	del. word	Deletes one word to the right of the cursor.
5	new word	If the cursor is on a non-blank character, this key deletes the word to the right and activates insert mode. If the cursor is on a space surrounded by non-blank characters, this key inserts a space and leaves the cursor in insert mode before the space. If the cursor is on a space surrounded by other spaces, *new word* has the same effect as *ins. mode*.
6	transpose	Exchanges the character at the cursor with the character to its right.
7	undo	Returns the command to its original (pre-edit) state.
8	done	Exits the history editor, erasing the command line.

Other keys also perform special command-editing functions while the command line editor is active. Table 8-2 lists those keys.

Table 8-2
Other Command Editing Keys

Key	Command Editing Function
CTRL-U	Moves the cursor to the leftmost character of the command on display.
CTRL-W	Moves the cursor left one word. A word is a series of non-blank characters.
BACKSPACE	Moves the cursor left one character without deleting the intervening character.
RUBOUT	Replaces the character at the cursor position with a space, and moves the cursor right one character.
Space	Ordinarily, moves the cursor right one character. If you are in insert mode, the space bar inserts spaces.
TAB	Moves the cursor one word to the right.
RETURN	Executes the current command line.

Saving Key History

When you exit Keyshell, Keyshell asks you whether you want to save the key history accumulated during the current session. If you answer "yes", Keyshell stores the key history in a set of "session files". In later Keyshell sessions, the preserved key history is restored for further use.

Note that session files save the command argument information from key labels; they do not preserve the command history accessed by the shifted function keys *History Fwd* and *History Back*.

Section 9
STANDARD TNIX COMMANDS

Section 9

STANDARD TNIX COMMANDS

INTRODUCTION

This section provides a brief description of most standard TNIX commands. The following Command Index lists the commands by function. The remainder of this section lists the commands alphabetically. For a complete description of any of these commands, enter the **man** command plus the command name. For example, **man cd** displays information about the **cd** command.

For information about any command that is not in the standard TNIX command set, use the **man** command to view the online manual page. You can also refer to the user's manual for the optional product that includes the command.

COMMAND INDEX

File and Directory Manipulation

Showing Files

File Processing

Program Development

Protection and Access

Communication

Status

Help

Terminal and Device Setup

Data Transfer

TNIX Command Language

System Manager Commands

(These system maintenance commands are discussed in the *8560 Series MUSDU System Manager's Guide.*)

Special Keys

CTRL-C	Interrupt command or program execution
CTRL-D	Signal the end of input (as in **mail**) or terminate the current subshell
CTRL-H	Delete input character
CTRL-K	Restore last line entered, one character for each CTRL-K
CTRL-Q	Resume output
CTRL-R	Reprint input line
CTRL-S	Suspend output
CTRL-U	Delete input line
BACKSPACE	Delete input character

TNIX Environment Variables

IU	Currently selected port number connected to 8540 Integration Unit or 8550 MDL
uP	Currently selected target microprocessor
HOME	Your login directory
PATH	Sequence of directories in which **sh** looks for commands
TERM	Currently selected terminal type
TERMCAP	Current definition of terminal capabilities
KSH	Current invocation flags and options for **ksh** command
MORE	Current invocation flags and options for **more** command
MAIL	Pathname of incoming mail
PS1	Primary **sh** prompt string
PS2	Secondary **sh** prompt string

NOTATION CONVENTIONS

The following conventions are used in this section:

1. Boldface items are required.
2. Underlined or italicized items represent parameters for you to enter.
3. Brackets [ ] enclose optional items.
4. Stacked items show that you may choose no more than one of the stacked items.
5. Trailing dots indicate that additional parameters may be entered.

COMMANDS

8540 and 8550

8540 *commandline*
8550 *commandline*

Sends the specified command line to the 8540 or 8550.

asm

asm *objectfile listfile sourcefile1* [*sourcefile2* ...]

Invokes the TEKTRONIX Series B Assembler to translate source code into object code. To assemble for a specific microprocessor, you must set the "uP" shell variable equal to the name of the microprocessor. For example, to use the 8085 microprocessor with the **asm** command:

```
$ uP=8085; export uP
```

The proper assembler must be installed on your system in order for the **asm** command to work.

objectfile The file to receive the object code generated by the assembler.

listfile The file to receive the listing generated by the assembler.

sourcefile One or more files that contain the source code to be assembled by **asm**.

atobobj

atobobj [**-o** *outfile*] *infile*

Reads an object module (in Series A format) from input object file *infile*, converts the object module into Series B format, then writes the converted object module into an output object file.

-o *outfile* Specifies the filename of the output object file.

If you omit *outfile*, the output object file is given the same name as *infile*, except that the filename has the two characters **".B"** appended to it. In this case, the input filename may not be longer than 12 characters; otherwise, truncation will occur when the characters **".B"** are appended.

infile The filename of the input object file (contains an object module in Series A format).

cat

cat [**-u**] [*file*] ...

Displays each *file* in sequence on the standard output. If a *file* argument is not given, or if the argument is '-', **cat** reads from the standard input. Output is buffered in 512-byte blocks unless the standard output is a terminal.

-u Output is not buffered in 512-byte blocks.

cd

cd *[directory]*

Directory becomes the new working directory. You must have execute (search) permission in *directory*.

chgrp

chgrp *group file...*

Chgrp changes the group-ID of *file(s)* to *group*. The group may be either a decimal group-ID or a group name found in the group-ID file.

chmod

chmod *who±permissions* [,*who±permissions*]... *file...*

Changes the mode (access permission) of *file* according to the given *person* and *permissions*. For a description of what each permission allows you to do, see the *TNIX Operating System* section of this manual. (You may also specify absolute permission using octal numbers. For information about absolute permission, enter **man chmod**.)

who	The person or people whose permissions you are altering.		
	u	user (you)	
	g	group	
	o	others (not you or your group)	
	a	all (you, group, other)	
permissions	The kind of permission you are adding or removing.		
	r	read permission	(+r = add) (-r = remove)
	w	write permission	(+w = add) (-w = remove)
	x	execute permission	(+x = add) (-x = remove)

chown

chown *owner file...*

Changes the owner of *file(s)* to *owner*. The owner may be either a decimal user-ID or a login name found in the password file.

cmp

cmp [**-ls**] *file1 file2*

Compares the two files. If *file1* is '-', the standard input is read.

-l	Prints the byte number (decimal) and the differing bytes (octal) for each difference.
-s	Generates return codes only.

comm

comm [**-123**] *file1 file2*

Reads *file1* and *file2*, which should be ordered in ASCII collating sequence, and produces a three column output: lines only in *file1;* lines only in *file2;* and lines in both files. The filename "-" refers to standard input. Flags 1, 2, or 3 suppress printing of the corresponding column.

cp

cp *file1 file2*
cp *file... directory*
cp *directory1 directory2*

Copies *file1* onto *file2*. The mode and owner of *file2* are preserved if the file already existed; the mode of the source file is used otherwise. The second form copies one or more *files* into the *directory*; each file keeps its original filename. The third form copies the subtree with the root at *directory1* to *directory2*. *Directory2* must not be in the *directory1* subtree.

date

date *[dd-mmm-yy] [hh:mm[:ss]] [-t SSS] [-d DDD] [[-w [hh:mm]] [-e [hh:mm]]]*
date *[[yy]mmddhhmm[.ss]] [-t SSS] [-d DDD] [[-w [hh:mm]] [-e [hh:mm]]]*

With no argument, displays the current date and time. With an argument, sets the current date and time. The arguments are:

dd	The day number in the month.
mmm	The lowercase first three letters of the month.
mm	The month number or minute number.
yy	The last two digits of the year.
hh	The hour number (24 hour system).
ss	The (optional) second number.
-t SSS	Sets the system standard time-zone string. If this string is not set or is null, the system will print out GMT (Greenwich Mean Time) +/- HH for the time-zone, where HH represents hours west of GMT.
-d DDD	Sets the system daylight time-zone string. If this string is null or was never set, the system assumes that there is no daylight savings time.
-w hh:mm	Sets the hours and minutes west of Greenwich Mean Time that the time-zone strings represent if you are west of the Prime Meridian.
-e hh:mm	Sets the hours and minutes east of Greenwich Mean Time that the time-zone strings represent if you are east of the Prime Meridian.

df

df [*filesystem*] ...

Displays the number of free blocks available for file allocation on the *filesystem(s)*. If no file system is specified, the free space on each filesystem listed in the file */etc/checklist* is printed. (If */etc/checklist* cannot be read by **df**, the */dev/rhd0* filesystem will be used.)

diff

diff [**-befh**] *file1 file2*

Compares two files and displays lines that are different. A filename of "-" refers to standard input. If *file1 (file2)* is a directory, then a file in that directory whose filename is the same as the filename of *file2 (file1)* is used.

-b Ignores trailing blanks (spaces and tabs); other strings of blanks are compared equal.

-e Produces a script of **a, c,** and **d** commands for the editor **ed,** which will recreate *file2* from *file1*.

-f Produces a similar script to that produced by **-e,** not useful with **ed,** in the opposite order.

-h Does a fast, simplified job. It works only when changed stretches are short and well separated, but does work on files of unlimited length.

dsc50

dsc50 [**-mwdxtls**] [**-epbrvq**] [**-c** *path*] [**-o** *owner*] [**-n** *name*] [**-a** *n*] [**-f** *disk*] [*source... dest*]

Performs many DOS/50 functions on an 8550 disk. The operation to perform can be specified as a command line option, or in an interactive mode. **Dsc50** defaults to interactive mode. An operation is specified by one of the following letters.

-m Makes a new DOS/50 volume. Similar to **-w**, except that the 8550 disk is initialized (without being physically formatted) prior to copying the file(s).

-w Write. Copies the TNIX *source* file(s) to the 8550 *dest* file/directory. A *source* file name of "-" refers to standard input.

-d Deletes the 8550 *source* files.

-x	Extract. Copies the 8550 *source* file(s) to the TNIX *dest* file/directory. A *dest* file name of "-" refers to standard output.
-t	Table of contents. Displays a directory listing of the *source* file(s). If no *source* is specified, the current directory is used.
-l	Links the existing 8550 *source* file(s) to the 8550 *dest* file/directory.
-s	Displays the setup (the current state of all options) in the form of a **dsc50** command. Useful only in interactive mode.
-c path	Changes the current 8550 directory to *path*. This will be the directory relative to which all 8550 pathnames not beginning with a slash (/) are interpreted.
-o owner	Specifies the owner name to be given to anything created on the 8550 disk (files, directories, or the entire volume). The default owner is "NO.NAME".
-n name	Specifies the volume name to be given to the 8550 disk when it is created by the **-m** operation. The default name is "NO.NAME".
-a n	Specifies the number (amount) of file slots to be created on the 8550 disk by the **-m** operation. The default is 256 slots.
-f disk	Specifies a file to use as the 8550 disk instead of the default device, */dev/rfd0*.
-e	Exits. Terminates interactive mode after executing the given operation. Ignored if not in interactive mode.
-p	Prompt. Displays a "-" prompt for each command. Ignored if not in interactive mode. The default is no prompt.
-b	Binary transfer. Because the "end of line" character is different for TNIX and DOS/50 text files, some character translation must occur when a text file is copied to or from the disk. This translation is performed by **dsc50** on all files unless the **-b** option is specified.
-r	Specifies recursive action.
-v	Verbose. Displays the name of each file it treats preceded by the operation name.
-q	Query. Pauses before treating each file, types the operation name and the file name (as with **-v**), and awaits your response before proceeding.

du

du [**-a** | **-s**] [*file*] ...

Displays the number of blocks contained in all files (directories) within each specified directory or file. If *file* is missing, the current directory is used. If all arguments are missing, information is displayed about each directory only.

-a Displays information about each file.

-s Displays only the grand total of each file or directory.

echo

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

Echo displays its arguments after they have been processed by the shell. Arguments are separated by blanks and terminated by a newline.

-n No newline is added to the output. This flag must precede all other arguments in order to be recognised.

ed

ed [**-cpx**] [*file*]

Invokes **ed**, the standard TNIX text editor. (**Ed** is described in Section 5 of this manual.)

-c Suppresses the printing of character counts by **e**, **r**, and **w** commands.

-p Turn on prompt character (*).

-x Used only with the **crypt** command of the optional 8560 Series MUSDU Native Programming Package. An **x** command is simulated first to handle an encrypted file.

expr

expr *expression*

Evaluates arguments as an expression and displays the result on the standard output. Each operator or operand of the expression is a separate argument. The operators and keywords are listed below. The list is in order of increasing precedence, with equal precedence operators grouped.

expr ¦ *expr*	Yields the first *expr* if it is neither null nor '0', otherwise yields the second *expr*.
expr & *expr*	Yields the first *expr* if neither *expr* is null or '0', otherwise yields '0'.
expr relop expr	*Relop* (relational operator) is < <= = != >= or >. Yields '1' if the indicated comparison is true, '0' if false. The comparison is numeric if both *expr* are integers, otherwise lexicographic.
expr + *expr*	Adds the arguments.
expr - *expr*	Subtracts the arguments.
expr * *expr*	Multiplies the arguments.
expr / *expr*	*Divides the arguments.*
expr % *expr*	Remainder of the arguments.
expr : *expr*	The matching operator compares the first argument (which is a string) with the second argument which is a regular expression whose syntax is the same as that of **ed**. The \(...\) pattern symbols can be used to select a portion of the first argument. Otherwise, the matching operator yields the number of characters matched ('0' on failure).
(*expr*)	Parentheses for grouping.

false

false

Sets exit status to non-zero.

fbr

fbr –{**cruxtd**} [**–viw**] [**–l** *comment*] [**–m** *directory*] [**–f** *archive*] [**–**] [*file*] ...

Saves and restores directories and files on a floppy disk archive, preserving multiple links to the same file.

-c	Creates a new archive.
-r	Replace. The named files are written on the archive.
-u	Updates the archive.
-x	Extracts the named files from the archive to the file system.
-t	Table of Contents. Lists the names of the specified files.
-d	Deletes the named files from the archive.
-v	Verbose. Displays the function name and the filename.
-i	Notes errors reading the archive, but takes no action.
-w	Displays the function name and the file name (as with **v**), and awaits your response.
-l	Allows you to add a label (comment).
-m directory	Prepends *directory* to all files operated on.
-f archive	Uses *archive,* rather than the floppy disk, as the archive.

find

find *pathname ... expression*

Recursively descends the directory hierarchy for each pathname in the *pathname* list, seeking files that match an *expression* written using the syntax given below.

-name filename	True if the *filename* argument matches the current file name.
-perm onum	True if the file permission flags (bits 0777) exactly match the octal number *onum.*
-type x	True if the type of the file is *x,* where *x* is **b, c, d** or **f** for block special file, character special file, directory or plain file.
-links n	True if the file has *n* links.
-user uname	True if the file belongs to the user *uname* (login name or numeric user ID).

-group gname	True if the file belongs to group *gname* (group name or numeric group ID).
-size n	True if the file is *n* blocks long (512 bytes per block).
-inum n	True if the file has inode number *n*.
-atime n	True if the file has been accessed in *n* days.
-mtime n	True if the file has been modified in *n* days.
-exec command	True if the executed command returns a zero value as exit status.
-ok command	Like **-exec** except that the generated command is written on the standard output, then the standard input is read and the command executed only upon response **y.**
-print	Always true; causes the current pathname to be printed.
-newer file	True if the current file has been modified more recently than the argument *file*.

format

format [**-s**]

Writes the appropriate formatting information to prepare a flexible disk for subsequent data storage.

-s	Formats the disk single density.

grep

grep [**-vclnbshy**] ... [[**-e**] *expression*] [*file*] ...
egrep [**-vclnbsh**] ... [**-f** *file*] [[**-e**] *expression*] [*file*] ...
fgrep [**-vclnbshx**] ... [**-f** *file*] [*string*] [*file*]

Commands of the **grep** family search the input *files* (standard input default) for lines that match *expression* or *string*. **Grep** search items are limited regular expressions in the style of **ed**. **Egrep** search items are *extended* regular expressions. **Fgrep** is fast and compact; its search items are literal strings (contain no regular-expression characters).

-v	Displays all lines except those that match.
-c	Displays only a count of matching lines.

-l	Displays the names of files with matching lines (once), separated by newlines.
-n	Display includes the line number in the file that contains the match.
-b	Display includes the block number on which each line was found.
-s	Displays only an exit status.
-h	Display does not include filename headers with output lines.
-y	Lowercase letters in the pattern will also match uppercase letters in the input (**grep** only).
-e expression	Same as a simple *expression* argument, but useful when the *expression* begins with a "-".
-f file	Takes the regular expression (**egrep**) or string list (**fgrep**) from *file*.
-x	(Exact) Displays only lines that match in their entirety (**fgrep** only).

help

help

Displays information about TNIX online manual pages.

index

index *keyword...*

Shows a list of names of online manual pages that contain pertinent information about the *keyword* or combination of keywords.

kill

kill [*-signal*] *processid...*

Terminates (with signal 15) the specified processes. If a signal number preceded by "-" is given as the first argument, that signal is sent instead of terminate. Enter **man signal** for more information on signals.

ksh

ksh [-CcFf]

Invokes the Keyshell interface to TNIX.

-c or -C Compact. Removes or adds 2 lines from the label area of the Keyshell display, depending on the current setting of the KSH environment variable.

-f or -F Function keys. Removes or adds or a line to the Keyshell display that associates a number with the label for each function key, depending on the current setting of the KSH environment variable.

libgen

libgen [-c *commandfile***] [-d** *modulename***] [-h** *string***] [-i** *filespec***] [-l] [-n** *filespec***] [-o** *filespec***] [-r** *filespec***] [-v] [-x** *modulename [filespec]***]**

The Library Generator (**libgen**) is a general-purpose utility program used to create and maintain object module libraries for use with the linker.

-c commandfile Invokes a **libgen** command file containing a series of **libgen** command options.

-d modulename Deletes library module(s).

-h string Specifies the header for the new library.

-i filespec Inserts new module(s) into the library.

-l Specifies listing options.

-n filespec Designates a new library file.

-o filespec Specifies an old library file.

-r filespec Replaces old module(s) with new modules(s).

-v Specifies a detailed listing.

-x modulename [filespec]
Extracts (copies) a module to an object file.

link

link [**-CDLOcdmorstx**] *[parameters]*

Link merges one or more independently assembled object files into a load file, suitable for loading into memory.

-C	Class definition command—assigns classname to section(s).
-D	Define symbol command—defines a global symbol at **link** time.
-L	Locate command option—locates class/section to a specified memory area.
-O	Object file command option—specifies object module, library, and *linked* load files to be *linked* by **link**.
-c	Command file command option—invokes a *linker* command file.
-d	Debug command option—triggers debug information.
-m	Memory map command option—defines a memory map configuration.
-o	Output load file command option—designates the output file for the *linked* code.
-r	Relink command option—triggers relink information.
-s	Symfile (global symbol file) command option—specifies symfile(s) to be *linked*.
-t	Type command option—specifies relocation type of named class(es) or section(s).
-x	Transfer address command option—specifies load module transfer address.

ln

ln *name1* [*name2*]

Ln creates a link to an existing file *name1*. If *name2* is given, the link has that name; otherwise it is placed in the current directory and its name is the last component of *name1*.

login

login [*username*]

If **login** is invoked without an argument, it asks for a user name, and, if appropriate, a password. Echoing is turned off (if possible) while you type the password, so it will not appear on the written record of the session. A helpful message is displayed after three unsuccessful attempts to log in.

lp1r and lp2r

lp1r [**-cmr**] [*file*] ...
lp2r [**-cmr**] [*file*] ...

Queues *files* for printing on line printer 1 (**lp1r**) or 2 (**lp2r**). If no files are named, or if the file "-" is encountered, the standard input is read.

-c	Copies the file, to insulate against changes that may happen before printing.
-m	Report by **mail** when printing is complete.
-r	Deletes the file once it has been queued.

ls, lf, ll, lr, lx

ls [*directory*]...
lf [*directory*]...
ll [*directory*]...
lr [*directory*]...
lx [*directory*]...

For each directory, **ls** and the related listing commands **ll**, **lf**, **lr**, **lx** list the contents of the directory.

There are many more ways to list the contents of directories. For full information, enter **man ls**. The output is sorted alphabetically by default.

ls	Lists contents in short format, multi-column, names sorted alphabetically down the columns.
lf	Marks directories with a trailing /, executable files with a trailing *, regular files with no mark.
ll	Lists contents in long format, giving access permissions, number of links, owner, size in bytes, and time of last modification for each file in *directory*.
lr	Lists *directory*, also recursively lists any subdirectories encountered.
lx	Lists in multi-column format, names sorted alphabetically across rows.

lstr

lstr [**–ghnosuv**] [*file* ...]

Lstr prints the symbol table of each object module in the argument list. If an argument is a library, a listing for each object file in the library is produced.

-g	Prints only global (external) symbols.
-h	Prints the header in a library inserted with the **libgen –h** switch.
-n	Sorts numerically rather than alphabetically.
-o	Prepends file or library element name to each output line rather than only once.
-s	Appends the length of sections to their output lines.
-u	Prints only undefined symbols.
-v	Prints the version number of **lstr**.

mail

mail *username...*
mail [**–rqp**] [**–f** *file*]

Mail with no argument displays your mail, message-by-message, in last-in, first-out order.

-r	Displays the contents of the mailbox in first-in, first-out order.
-q	Causes **mail** to exit after interrupts without changing the mailbox, rather than just stopping printing of the current letter.
-p	Prints contents of the mailbox without questions.
-f file	Displays the given file as if it was the mail file. For each message, **mail** reads a line from the standard input to direct disposition of the message.

Once you have invoked **mail**, you can use the following **mail** commands to process your messages as you read them:

RETURN	Goes on to next message.
d	Deletes message and goes on to the next.
p	Prints message again.
-	Goes back to previous message.

s [file] ...	Saves the message in the named *files* ('mbox' default).
w [file] ...	Saves the message, without a header, in the named *files* ('mbox' default).
m [username] ...	Mails the message to the named *usernames* (yourself is default).
CTRL-D	Puts unexamined mail back in the mailbox and stops.
q	Same as CTRL-D.
x	Exits, without changing the mailbox file.
! command	Escapes to the shell to do *command*.
?	Prints a summary of **mail** commands.

make

make [**-ikntrs**][**-f** *makefile*][*file*]...

Make executes commands in *makefile* to update one or more target *files*. The default is to use the script in a file named *makefile* in the current directory (or *Makefile* if *makefile* does not exist).

-i	Equivalent to the special entry '.IGNORE:'.
-k	When a command returns nonzero status, abandons work on the current entry, but continues on branches that do not depend on the current entry.
-n	Traces and prints, but does not execute the commands needed to update the targets.
-t	Touches, i.e. updates the modified date of targets, without executing any commands.
-r	Equivalent to an initial special entry '.SUFFIXES:' with no list.
-s	Equivalent to the special entry '.SILENT:'.
-f makefile	The given file is used as the script rather than *makefile* or *Makefile*. If *makefile* is "-", the standard input is used as the script. More than one **-f** option may appear.

man

man *title* ...

Man displays the online manual page named *title*, showing a screenful at a time.

mesg

mesg [**-ny**]

Reports the current state of message permission.

-n Prevents other users from communicating with you via **write.**

-y Allows other users to communicate with you via **write.**

mkdir

mkdir *dirname ...*

Creates the specified directory in absolute mode 0777 (readable, writeable, searchable by everyone).

mload

mload [**-ds** / **-us**] [**-p** *cc...c*] *name*

Provides the host-side protocol for formatted data transfers to or from TNIX.

-u Specifies upload

-d Specifies download (the default).

-p cc...c An optional prompt, corresponding to the *P=* parameter on the TEKDOS COMM command.

-s Slows down the transfer. A slowed transfer is required by TEKDOS on the 8002.

more

more [-d]

Limits lines of output coming to the screen to one screenful at a time.

-d Prompts with the message "Hit space to continue, Rubout to abort" at the end of each screenful.

mv

mv *file1 file2*
mv *file... directory*
mv *directory1 directory2*

Moves (changes the name of) *file1* to *file2*. In the second form, one or more *files* are moved to the *directory*; they retain their original filenames. In the third form, the subtree rooted at *directory1* is moved to *directory2*.

newgrp

newgrp *group*

Temporarily changes your group identification.

nice

nice [*-number*] *command*
nohup *command*

Nice executes *command* with low scheduling priority. **Nohup** executes *command* even if you logout or terminate a command from your terminal. The priority for **nohup** is incremented by 5.

-number — Increments the priority by *number* up to 20 (higher numbers mean lower priorities). *Number* defaults to 10.

od

od [**-bcdox**] [**-s** *offset*[.][**b**]] [**-e** *offset*[.][**b**]] [*file*]

Od dumps *file* in one or more formats as selected by the flags. Defaults to **-o**.

-b — Interprets bytes in octal.

-c — Interprets bytes in ASCII. Certain non-graphic characters appear as C escapes: null=\0, backspace=\b, formfeed=\f, newline=\n, return=\r, tab=\t; others appear as 3-digit octal numbers.

-d — Interprets words in decimal.

-o — Interprets words in octal.

-x Interprets words in hex.

-s offset Specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If "." is appended, the offset is interpreted in decimal. If "b" is appended, the offset is interpreted in blocks of 512 bytes.

-e offset Same as **-s**, but specifies where dumping is to end. Default is dumping until end-of-file.

passwd

passwd [*name*]

Changes or installs a password associated with the user *name* (your own name by default).

pr

pr [**-ntm**] [**-h** *header*] [**-c** *n*] [**-p** *n*] [**-w** *n*] [**-l** *n*] [**-s** *c*] [*file*] ...

Produces a listing, formatted for the printer, of one or more *files*. The output is separated into pages headed by a date, the name of the file or a specified header, and the page number. If there are no file arguments, **pr** prints its standard input.

n Adds line numbers.

-t Does not print the 5-line header or the 5-line trailer normally supplied for each page.

-m Prints all *files* simultaneously, each in one column.

-h header Uses *header* as the page header.

-c n Produces *n*-column output.

-p n Begins printing with page *n*.

-w n For multi-column output, sets the width of the page to *n* characters instead of the default 72.

-l n Sets the length of the page to be *n* lines instead of the default 66.

-s c Separates columns by the single character *c* instead of by the appropriate amount of white space.

ps

ps [-axl]

Displays information about active commands (processes).

-a Displays information about all processes rather than only one's own processes.

-x Also displays information about processes not necessarily connected with a terminal.

-l Produces a long listing. The short (default) listing contains the process ID, terminal number, cumulative execution time of the process and an approximation of the command line. For an explanation of the long listing, enter **man ps**.

pwd

pwd

Displays the pathname of the working (current) directory.

rm

rm [-fri] *file...*

Removes the entries for one or more *file(s)* from a directory.

-f Forces a file with no write permission to be deleted without question.

-r Recursively deletes the contents of the specified directory, including the directory itself.

-i Interactive: asks whether to delete each file, and, under **-r** whether to examine each directory.

rmdir

rmdir *dir...*

Removes entries for the named directories, which must be empty.

settape

settape *name...* [**–q**] [**–n** / **–r**]

Prepares the specified magnetic tape device for use and displays the device's characteristics.

Settape always resets all the system characteristics to their default values (which are contained in the */etc/gpib/dev* file). **Settape** modifies the operational characteristics only when you specify **–n** or **–r**.

name	The complete pathname of the magnetic tape device. The pathname must always begin with */dev/*. If you include more than one *name*, the **–n**, **–q**, and **–r** options apply to each device. **Settape** checks each *name* before modifying any characteristics, to be sure that *name* specifies a GPIB-compatible device.
-q	Quiet: **settape** does not print out device characteristics.
-n	Specifies that *name* does not rewind the tape when closed. The **–n** and **–r** options are mutually exclusive.
-r	Specifies that *name* rewinds the tape when closed. The **–r** and **–n** options are mutually exclusive.

sh

sh [**–csientuvx**] [*arg*] ...

Executes commands read from a terminal or a file.

-c string	Commands are read from *string*.
-s	If the **–s** flag is present or if no arguments remain, then commands are read from the standard input. Shell output is written to file descriptor 2 (standard output).
-i	If the **–i** flag is present or if the shell input and output are attached to a terminal, then this shell is *interactive*.
-e	If non-interactive, the shell exits immediately if a command fails.
-n	Reads commands but does not execute them.
-t	Exits after reading and executing one command.
-u	Treats unset variables as an error when substituting.

-v Displays shell input lines as they are read.

-x Displays commands and their arguments as they are executed.

- Disables the **-x** and **-v** options.

sleep

sleep *time*

Suspends execution for *time* seconds.

slp

slp *printer options...*

Modifies the output characteristics of *printer* according to *options*, exiting with a status of 2 if the command was invoked improperly, 1 if some other problem occurred, or 0 if all went well. *Printer* is the special file for the desired printer (e.g. */dev/lp1*).

nl The attached printer processes newlines—**slp** performs no newline processing.

-nl The attached printer does not process newlines—**slp** replaces newline with carriage return/linefeed on output.

l=string Replaces newline on output with *string*.

tabs The attached printer processes tabs—**slp** performs no tab processing.

-tabs The attached printer does not process tabs—**slp** replaces tabs with the appropriate number of spaces on output.

sort

sort [**-bdfinrcmu**] [**-t** *c*] [*+pos1* [*-pos2*]] ... [**-o** *file*] [**-T** *directory*] [*file*] ...

Sorts lines of all the named files together and writes the result on the standard output.

-b Ignores leading blanks (spaces and tabs) in field comparisons.

-d 'Dictionary' order: only letters, digits and blanks are significant in comparisons.

-f Folds uppercase letters onto lowercase.

-i Ignores characters outside the ASCII range 040-0176 in nonnumeric comparisons.

-n	An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value.
-r	Reverse the sense of comparisons.
-c	Checks that the input file is sorted according to the ordering rules; gives no output unless the file is out of sort.
-m	Merges two already sorted input files.
-u	Suppresses all but one in each set of equal lines. Ignored bytes and bytes outside keys are omitted from this comparison.
-t c	'Tab character' separating fields is *c*.
-o file	Specifies the name of an output file to use instead of the standard output. This file may be the same as one of the inputs.
-T directory	Specifies the directory in which temporary files should be placed.

statgpib

statgpib [*device* ...]

Displays the status of GPIB-compatible devices on the system.

device	The complete pathname of a GPIB-compatible device. The pathname must always begin with */dev/*. You may specify several devices with one **statgpib** command.

stty [*option*] ...

Sets certain I/O options on the current output terminal. With no argument, it reports the current settings of the options. An appended "i" refers to input, an "o" refers to output.

f	Forces the specified change to occur even if the terminal characteristic table has been changed.
even{io}	Even parity on input, output
odd{io}	Odd parity on input, output
nopar{io}	No parity(0) on input, output

mark{io}	Mark parity(1) on input, output
data{io}	Data on input,output (8th bit is not stripped)
nocare{io}	Don't-care parity on input, output
raw	Raw mode input (no erase, kill, interrupt, quit, EOT; parity bit passed back)
-raw	Negates raw mode
cooked	Same as '-raw'
cbreak	Makes each character available to **read** as received; no erase and kill
-cbreak	Makes characters available to **read** only when newline is received
-nl	Allows carriage return for newline, and output CR-LF for carriage return or newline
nl	Accepts only newline to end lines
echo	Echoes every character typed
-echo	Does not echo characters
aresm	Turns on auto resume, which causes any input to resume the output if it has been suspended.
-aresm	Turns off auto resume.
xonxof	Turns on XON/XOF flagging on input.
-xonxof	Turns off XON/XOF flagging on input.
-tabs	Replaces tabs by spaces when printing
tabs	Preserves tabs
ek	Resets erase and kill characters back to normal ^H and ^U
erase c	Sets erase character to *c*. *C* can be of the form '^*X*' which is interpreted as a 'control X'.
kill c	Sets kill character to *c*.
300 600 1200 2400 4800 9600	Sets terminal baud rate to the number given, if possible.
IU	Specifies that an Integration Unit (IU) is connected to the port.

su

su [*username*]

Requests the password of the specified *username*. If it is given, **su** changes to that *username* and invokes the shell without changing the current directory or the user environment. Entering CTRL-D returns to the previous shell and username.

sync

sync

Executes the **sync** system call.

tail

tail [**-rlbc**][**-s** *number*][**-e** *number*][*file*]

Copies the named file to the standard output beginning at a designated place.

-r	Prints lines in reversed order.
-l	*Number* is in number of lines.
-b	*Number* is in number of 512-byte blocks.
-c	*Number* is in number of characters.
-s number	Prints beginning at *number* units from the start of the file. Units defaults to lines.
-e number	Prints beginning at *number* units from the end of the file. Units defaults to lines.

tee

tee [**-ia**] [*file*] ...

Makes a copy of the standard input to *files*.

-i	Interrupts are ignored.
-a	Output is appended to the *files* rather than overwriting the files.

test

test *expr*

Evaluates the expression *expr,* and if its value is true then returns zero exit status. If the value of *expr* turns out to be false, a non zero exit status is returned. The following elements are used to construct the *expr.*

-r file	True if the file exists and is readable.
-w file	True if the file exists and is writable.
-f file	True if the file exists and is not a directory.
-d file	True if the file exists and is a directory.
-s file	True if the file exists and has a size greater than zero.
-t [fildes]	True if the open file whose file descriptor number is *fildes* (1 by default) is associated with a terminal device.
-z s1	True if the length of string *s1* is zero.
-n s1	True if the length of the string *s1* is nonzero.
s1 = s2	True if the strings *s1* and *s2* are equal.
s1 != s2	True if the strings *s1* and *s2* are not equal.
s1	True if *s1* is not the null string.
n1 -eq n2	True if the integers *n1* and *n2* are algebraically equal. Any of the comparisons **-ne, -gt, -ge, -lt,** or **-le** may be used in place of **-eq.**
!	Unary negation operator
-a	Binary **and** operator
-o	Binary **or** operator
(expr)	Parentheses for grouping.

time

time *command*

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

touch

touch [**-c**] *file* ...

Attempts to set the modified date of each *file*.

-c Does not create a file.

tr

tr [**-cds**] [*string1*] [*string2*]

Copies the standard input to the standard output with substitution or deletion of selected characters.

-c Complements the set of characters in *string1* with respect to the universe of characters whose ASCII codes are 01 through 0377 octal.

-d Deletes all input characters in *string1*.

-s Squeezes all strings of repeated output characters that are in *string2*.

true

true
false

True sets exit status to zero. **False** sets exit status to non-zero.

tset

tset [**-sQ**] [?]*type*

Sets characteristics according to your terminal *type*: delays, baud rates, erase and kill characters, and so on. Enter **man tset** for more options to this command.

-s Executes TNIX environment variable assignments and exports the variables.

-Q Suppresses messages about the current settings of erase and kill characters.

? If a question mark appears just before the name for the terminal, you are queried as to whether you really want that type of terminal. Enter RETURN if *type* is correct; otherwise, enter another name.

type The name for your terminal, as used in the file */etc/termcap*.

tty

tty

Displays the pathname of your terminal.

uload

uload [**-d** | **-u**] [**-p** *cc...c*] [**-s** *nn*] *file*

Uload provides the host-side protocol for unformatted text transfers to or from TNIX.

d	Specifies download (the default).
u	Specifies upload.
-p cc...c	An optional prompt, corresponding to the *P=* parameter on the TEKDOS COMM command.
-s nn	Slows down the transfer by padding each line with nulls, where *nn* specifies the number of nulls to send. A slower transfer is needed for the 8002A.

uniq

uniq [**-udc** [*+n*] [*-n*]] [*inputfile* [*outputfile*]]

Reads the input file and compares adjacent lines.

-u	Outputs only the lines that are not repeated in the original file.
-d	Writes one copy of only the repeated lines.
-c	Supersedes **-u** and **-d** and generates an output report in default style but with each line preceded by a count of the number of times it occurred.
-n	The first *n* fields (together with any blanks before each) are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.
n	The first *n* characters are ignored. Fields are skipped before characters.

wait

wait

Waits until all background processes have completed, then reports on abnormal terminations.

wc

wc [**-lwc**] [*file*]...

Counts lines, words and characters in the named files, or in the standard input if no filename appears.

-l	Counts lines only.
-w	Counts words only.
-c	Counts characters only.

who

who [*file*]
who [**am I**]

With no argument, lists the login name, terminal name, and login time for each current TNIX user.

file	file is examined.
who am I	tells who you are logged in as.

write

write *user* [*ttyname*]

Copies lines from your terminal to that of another user. If you want to write to a user who is logged in more than once, you can use the *ttyname* argument to indicate the appropriate terminal name. Permission to write may be denied or granted by use of the **mesg** command. If the character "!" is found at the beginning of a line, **write** calls the shell to execute the rest of the line as a command. You terminate your messages by entering CTRL-D.

Section 10

ERROR MESSAGES

This section contains error messages and their explanations. These errors may be encountered when using the commands found in the standard package of the TNIX operating system commands. (Optional command packages may generate other error messages.)

The error messages are divided into two lists:

- *Shell Error Messages* are error messages that may occur while you are using any TNIX command.
- *Keyshell Error Messages* are those error messages that may occur only while you are using the Keyshell user interface.

If you encounter unfamiliar terms in the error message explanations, refer to the *Glossary* and the *Index* sections of this manual.

SHELL ERROR MESSAGES

'filename' and 'filename' are identical. Issued by **mv**. **Mv** will not move a file to itself.

'filename' does not exist. The pathname given was faulty, or the file does not exist.

'filename' exists. The file being moved to or created already exists.

'filename' is a directory. Issued by **ln** and **mv**. An attempt was made to operate on a directory as if it were a file.

'filename' is not a device or regular file. Issued by **dsc50** and **fbr**. The file specified as the DOS/50 or archive disk is of the wrong file type, such as a directory.

'filename' is not an object module. Make attempted to load a file that is not an object module.

'filename' not changed. You attempted to remove a file that you do not have write access to, so **rm** did not remove the file. Occurs when using the **–r** option of **rm**.

'filename' not in archive. The specified file does not exist or was not found on the flexible disk that **fbr** is operating upon.

'filename' not removed. Rm attempted to remove a file that (1) does not exist, (2) has a parent directory that cannot be written to, (3) is currently being executed, or (4) is itself a directory.

'string' busy; try again in a minute. Your mail box file (*mbox* by default), which is used to receive your mail, is currently being copied to. Wait until it is done and then try the **mail** command again.

'string' not remade because of errors. A program was not constructed because of errors in the **make** description file.

'username' logged more than once writing to 'tty number'. Issued by **write**. The person you are writing to is logged in on more than one terminal. **Write** tells you which terminal it is going to write to.

'username' not logged in. Issued by **write**. The person you are attempting to **write** to is not logged in.

) missing. An unmatched parenthesis was found in the command string.

] missing. An unmatched bracket was found in the command string.

abort after 'n' tries. Mload retries *n* times; if errors occur repeatedly in the transmission *n* times, the transmission is terminated.

alarm call. An *alarm* system call was sent from one process to a process that did not have a corresponding *signal* system call to receive the alarm. The receiving process terminates.

arg count. You specified the wrong number of arguments to a command.

arg list too long. You specified an argument list longer than 5120 bytes to a command.

argument expected. An argument was not found where it was expected in a command syntax.

argument too large. The value of an argument is too big.

argument too long. Issued by **grep**. The length of an argument string is greater than 256 characters.

bad address. The system encountered a hardware fault in attempting to access the arguments of a system call.

bad character 'char' (octal 'number'), line 'n'. Make has read an illegal character in a makefile.

bad directory <filename >. An attempt was made by **du** to change the current directory to ".." (the parent directory) within this directory.

bad directory tree. Issued by **find**. An attempt to change directories failed.

bad file number. Either a file descriptor does not refer to an open file, or a *read* (*write*) system call is made to a file that is open only for writing (reading).

bad free block 'block no'. Issued by **df**. One of the block numbers in the list of free blocks is out of range. The file system needs to be repaired with stand-alone **syschk**. See your 8560 system manager.

bad free count, b='block no.'. Issued by **df**. An incorrect number of free blocks was encountered. The file system needs to be repaired with stand-alone **syschk**. See your 8560 system manager.

bad number. Issued by **sh**. A string that is supposed to convert to a number could not be converted.

bad starting directory. Issued by **find**. You attempted to start searching at a location that (1) is not a directory, (2) you do not have read or execute access to, or (3) does not exist.

bad status < filename >. The *stat* system call, executed automatically by the **find** or **du** command, indicates that *filename* does not exist.

bad substitution. The shell encountered an illegal argument when it attempted to expand an expression.

bad system call. A call to the system failed.

bad trap. Issued by **sh**. The system call *signal*, used in a process, received a signal from another process, but the signal number received was illegal.

block device required. A block I/O file was required, but a non-I/O file (not an I/O device) or an I/O file (indicating an I/O device) that cannot perform block I/O, was specified.

broken pipe. An attempt was made to write to a pipe without another process to receive the pipe's output. This condition normally generates a signal; the error is returned if the receiving process's *signal* system call did not catch the sent signal.

bus error. An attempt was made to address out-of-bounds memory.

can check only 1 file. An attempt was made to use **sort –c** on more than one file at a time.

cannot access 'filename'. The file does not exist or exists in a directory which is not accessible to you.

cannot append to 'filename'. Mail failed while attempting to open (using the standard subroutine *fopen*) the mail file for appending.

cannot archive a link to the archive, 'filename'. Fbr prohibits attempts to archive a file into itself. The file either does not exist or is not a directory.

cannot change directory to /dev. Ps attempted to change to the *dev* directory, but the attempt failed. Enter **man chdir** for information on the *chdir* system call.

cannot change DOS/50 directory to 'filename'. Issued by **dsc50**. You cannot change to the given DOS/50 directory because it either does not exist or it is not a directory.

cannot change mode. The **mesg** command tried to change the write access bit on a terminal to permit message sending. You do not have write access to the terminal.

cannot change mode on 'filename'. Issued by **chmod** and **mesg**. An attempt was made to change the mode of a file which you did not own.

cannot change owner on 'filename'. If the user restoring files from a **fbr** disk is a superuser (logged in as "root"), **fbr** attempts to restore the files to their original owners. This message indicates that the attempt to restore the files to their original owners failed.

cannot chdir( ). While **du** was traversing a tree, a call to the system call *chdir* returned an error.

cannot copy file to itself. The source file and the destination file must be different for the **cp** command.

cannot copy from 'filename'. The file you want to copy from is read-protected.

cannot copy multiple files to a non-directory. Cp cannot copy multiple files to a single file. If you wish to concatenate several files, use **cat.**

cannot copy to 'filename'. The file you want to copy to is write-protected.

cannot create 'filename'. There are six possible causes for this error: (1) A needed directory cannot be searched (you lack execute permission), (2) the file does not exist and the directory cannot be written to, (3) the file is a directory, (4) the file exists but cannot be written to, (5) there are already too many files open or (6) the **passwd** or **diff** commands attempted to create a temporary file in a directory that did not exist (probably */usr/tmp* or */tmp*).

cannot create proc for remote. In an attempt to mail a letter to another TNIX system, **mail** was unable to call **uux**, the utility that handles remote mail. The creation of the process to execute **uux** failed. Enter **man fork** for information on the creation of processes.

cannot create process, try again. Mv tried to create a process when there was temporarily no room to create any more processes. Try the command again.

cannot exec cp. Issued by **mv**. The invocation of **cp** (automatically done by **mv**) to copy files failed. Contact your Tektronix field service representative.

cannot exec mkdir. Issued by **dsc50** and **fbr**. The program could not invoke **mkdir**. Contact your Tektronix field service representative.

cannot execute. Your attempt to execute a program failed. The file may not be executable, or you may not have permission to execute the file.

cannot extend a contiguous file. Issued by **dsc50**. The specified file's allocation type is "contiguous," meaning that its data must occupy adjacent blocks on the disk. The allocation for a contiguous file cannot be changed.

cannot find daemon. files left in spooling dir. The line printer daemon, an internal program, terminated. This daemon program takes care of the line printer queue, selecting the files to print. Try to print the file again.

cannot find diffh. A file is too big for the **diff** command. The **diffh** command, automatically called to handle these larger files, is not installed or is not in a standard location. Contact your Tektronix field service representative.

cannot find groupname 'string'. The specified group was not found in the group file.

cannot find rmdir. Issued by **rm**. The **rmdir** command is not installed or is not in a standard location. Contact your Tektronix field service representative.

cannot find username 'string'. The specified user name was not found in the password file.

cannot find your tty. The **write** command called the system call *ttyname* to find some information about your terminal, and *ttyname* returned null. Enter **man ttyname** for more information.

cannot generate key. The **crypt** command called the system call *read* to read in the key (password) from the command line, but the *read* failed.

cannot get current DOS/50 directory. Issued by **dsc50**. Due to corruption of your DOS/50 disk, your current location on the disk cannot be converted into a legal pathname.

cannot get disk size. Issued by **syschk**. A call to the system call *ioctl* to find the size of a file system failed, probably because the wrong device (special file) is being checked (for instance, the special file */dev/hd0* instead of */dev/rhd0*).

cannot link 'filename'. Mkdir failed to create a directory. **Mkdir** functions by creating a link between the ".." file in the new directory, and the parent directory of the new directory. This link failed. Enter **man ln** for more information.

cannot link 'filename1' to 'filename2'. Either the files are on different devices, or there can be no more links to that file, or you do not have write access to the directory that will contain the new link.

cannot load 'command'. Make failed to invoke a command. Check your **make** file.

cannot locate parent. When **mv** called the system call *stat* to determine some information about the parent of the current directory, the *stat* call failed.

cannot locate temp. Sort attempted to create a temporary file under */usr/tmp* or */tmp*, but could not find those directories.

cannot make directory 'filename'. You do not have write permission in current directory, or the file already exists and is write-protected.

cannot make pipe. Issued by **sh**. There were already too many files open when the pipe was attempted, and thus the shell could not open another input or output file. Wait until some of your open files are closed and then issue the command again.

cannot mount 'filename'. Issued by **dsc50**. The specified file is not in DOS/50 format.

cannot move a directory into itself. Issued by **mv**. A directory cannot become one of its own subtrees.

cannot move bad block 'block number' to bad block i-node. Issued by **syschk**. The bad block i-node (an internal data structure) cannot be extended to accommodate the specified bad block. Make sure that there are enough free blocks left on the file system.

cannot move bad blocks—cannot get bad block i-node. Near the end of the process of repairing a file system, **syschk** allocates all bad blocks to an i-node (an internal data structure) reserved for that purpose. This error message appears if the bad-block i-node cannot be read.

cannot move directories across devices. Issued by **mv**. Links between files in different logical devices are not allowed.

cannot open 'filename'. Either the file cannot be found, or you do not have read permission.

cannot open 'filename' for writing. Issued by **mail** and **uload**. Either the file cannot be found, or you do not have write permission.

cannot overwrite 'filename'. Issued by **cp**. You do not have write permission for the file that you are trying to copy to.

cannot read 'filename'. You do not have read permission for the file.

cannot read from non-8560 standard-in. A **dsc50** extract operation was specified, but a hyphen (-) was specified as the DOS/50 source file.

cannot recreate passwd file. Passwd failed to recreate the */etc/passwd* file. To recreate the file, **passwd** must have write access to the directory */etc*.

cannot remove 'filename'. Lpr failed to remove the file when the **–r** option was used, or you tried to use **rm** to remove the ".." directory.

cannot remove directory 'filename'. Issued by **dsc50**. An attempt to delete a DOS/50 directory failed, probably because the directory was not empty.

cannot rename 'filename'. Issued by **lpr** and **mv**. You do not have write permission on the directory that contains the file you tried to rename.

cannot reopen 'filename' for reading. Issued by **mail** and **dsc50**. An attempt was made to open a file that was already open.

cannot re-read 'filename'. Mail failed in an attempt to re-open a file for reading. Enter **man fopen** for more information.

cannot rewrite 'filename'. Mail failed to open the file for writing.

cannot seek on 'filename'. Issued by **od**. The call to the standard subroutine *fseek* (used by the **od** command) failed. Enter **man fseek** for more information.

cannot send to 'username'. The person you are sending mail to is probably not set up to receive mail.

cannot shift. A **shift** shell command failed, possibly because there were too few arguments on the command line.

cannot spare block 'block number'. Issued by **syschk**. The specified bad block could not be replaced with a good disk block. A bad block may have already been replaced on the spare sector in the bad block's track.

cannot touch 'filename'. Issued by **make** or **touch**. You cannot either open, read, or write to the file.

cannot unlink 'filename'. You do not have write permission for the file or for the directory that contains the file.

cannot update times on 'filename'. Fbr failed in an attempt to restore the modification and access times of the files.

cannot use wildcards when adding to archive. When performing any operation except extracting, **fbr** accepts filename expressions of the form acceptable to the shell. With the 'extract' operation, however, wildcard characters (also called metacharacters) are not allowed.

cannot write to non-8560 standard-out. Issued by **dsc50**. A replace or update command was given, but a hyphen (-) was specified as the DOS/50 destination file.

command terminated abnormally. Issued by **time**. The program you were timing exited with a non-zero status. Enter **man exit** for more information.

copy file 'filename' is too large. A file copied by the **lpr** command must be no larger than 204,800 bytes.

core dumped. The system detected a program failure. A core dump (the contents of the main memory segment that contained the program) is placed in a file called *core*. The core dump is the binary executable code at the time the program failed.

corrupted archive: bad archive label. The archive label (the part of the **fbr** disk that describes how the rest of the disk is organized) is incorrect. This usually indicates that the data has been improperly overwritten.

corrupted archive: bad directory checksum. A checksum is calculated for all directory information on a **fbr** disk. This message usually indicates that the data has been improperly overwritten.

corrupted archive: files partially overlap. The allocation on the **fbr** disk is no longer correct. This usually indicates that the data has been improperly overwritten.

could not create special file 'filename'. Issued by **cp**. A special file is a device and cannot be copied.

cross-device link. You may not link to a file that is in a file system on another logical device.

description file error. The **make** description file contained an error. Refer to the *Maintaining Files* section of this manual for information on the **make** command.

directory 'filename' is unreadable. The files in this directory were not copied because you do not have read access for the directory.

directory rename only. Issued by **mv**. The directory cannot be moved; it can only be renamed in place. This error may occur if the protection modes of the receiving directory will not permit write access.

don't know how to make 'program name'. Issued by **make**. Your makefile does not contain sufficient information to make your program.

DOS/50 file already exists. Issued by **dsc50**. An existing DOS/50 file was mentioned in an inappropriate context, such as linking.

DOS/50 file too fragmented. Issued by **dsc50**. The specified DOS/50 file cannot be extended because the DOS/50 disk free space is too fragmented. Delete some DOS/50 files or copy your files to a fresh DOS/50 disk.

DOS/50 permission denied. An attempt was made to access a DOS/50 file in a way forbidden by **dsc50**.

EMT trap. A program you invoked issued an illegal EMT assembly language trap instruction.

EOF on 'filename'. This message indicates an unexpected end-of-file while reading a file.

***** Error code 'decimal number'.** An error occurred during execution of the **make** command.

error writing control file. An error occurred while the **lpr** command was attempting to write to a control file it was using.

exec format error. A request was made to execute an object file which, although it has the appropriate permission, does not start with a valid format. Enter **man a.out** for more information on the standard executable file used by TNIX.

extra flag '–(char)'. An incorrect option was entered. That is, an option which has no meaning to the current command, or a correct option entered twice.

extra string argument. An extra filename was specified in a **dsc50** command.

fatal read error on 'filename'. A read error occurred on **syschk**'s temporary file.

fatal seek error on 'filename'. An I/O error occurred on **syschk**'s temporary file. An *lseek* system call failed.

fatal short read on 'filename'. An I/O error occurred on **syschk**'s temporary file. An *lseek* system call failed.

fatal short write on 'filename'. An I/O error occurred on **syschk**'s temporary file. An *lseek* system call failed.

fatal write error on 'filename'. An I/O error occurred on **syschk**'s temporary file. An *lseek* system call failed.

file 'filename' is empty. An I/O error occurred on **syschk**'s temporary file. The file you have sent to the printer is empty.

file changed after being used. A file used by **make** changed while **make** was running.

file exists. An existing file was mentioned in an inappropriate context. For example, an attempt to form a link (with **ln**) to a file that already exists can produce this message.

file table overflow. No more files can be opened until one or more files have been closed.

file too large. A file exceeded the maximum size (roughly 1000 megabytes).

files too big, try –h. The files you are trying to compare are too large for **diff**. Try using the **–h** option.

floating exception. Your program performed some floating point arithmetic operations that caused a floating point error.

hangup. Issued by **sh**. The hangup signal was issued to a process, but the process was not prepared to catch it. Enter **man signal** for more information.

I/O error. A physical I/O error occurred during a *read* or *write* system call. This error may occur on a system call following the one that actually caused the error.

ignoring standalone flag 'option'. A command option was selected that is not available in the command version of **syschk**. Use the stand-alone **syschk**.

Illegal DOS/50 file descriptor. An internal **dsc50** error occurred. Contact your Tektronix field service representative.

Illegal DOS/50 filename. Issued by **dsc50**. The specified DOS/50 filename contained characters not allowed by DOS/50.

Illegal DOS/50 seek. An internal **dsc50** error occurred. Contact your Tektronix field service representative.

illegal instruction. A program attempted to execute an illegal assembly language instruction, such as trying to execute data. Check your source code.

illegal io. An attempt was made to write to a write-protected device.

illegal seek. An *lseek* system call was issued to a pipe or to a device that does not implement *seek* operations. Enter **man seek** for more information.

incomplete –'option' option. Issued by **lpr**. This option requires additional information.

incomplete statement. Issued by **find**. Check your statement syntax.

inconsistent rules lines for 'string'. Check your **make** file. Refer to the *Maintaining Files* section in this manual.

input 'filename' is output. Your file will be destroyed if you use the **cat** command to move the file into itself.

input not tty. Issued by **tty**. The file you identified as a terminal device is not really a terminal device.

interrupted system call. An asynchronous signal (such as interrupt or quit), which you have elected to receive with a *signal* system call, occurred during the execution of another system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

invalid argument. This error may be caused by using an unknown signal number with the *signal* system call, by reading or writing a file for which the system call *lseek* has generated a negative pointer, or by math functions. Enter **man signal** or **man lseek** for more information.

invalid command. Mail was given an invalid mail command.

Invalid DOS/50 parameter. An internal **dsc50** error occurred. Contact your Tektronix field service representative.

invalid mode. Issued by **chmod**. You are trying to change the file to an invalid protection mode. Enter **man chmod** for information on protection modes.

invalid user id. Lpr cannot find the user ID in the */etc/passwd* file.

IOT trap. A program you invoked issued an illegal assembly language IOT trap instruction. Check your source code.

Is a DOS/50 directory. Issued by **dsc50**. A directory was specified where a DOS/50 file should have been specified.

Is a non-empty DOS/50 directory. Issued by **dsc50**. An attempt was made to remove a DOS/50 directory that was not empty.

is not an identifier. Issued by **sh**. An identifier was expected.

is read only. The file or device cannot be written to or executed, only read.

jackpot, you may have an unnecessary change recorded. The **diff** command made an error and said that there was a difference between two files when there was no difference.

keynotes: ... If you receive an error message beginning with "keynotes:", refer to the Keyshell Error Messages later in this section.

kill: 'process ID'—No such process. You attempted to use the **kill** command to terminate a process that does not exist.

killed. Issued by **sh**. One of your processes was terminated.

kpp: ... If you receive an error message beginning with "kpp:", refer to the Keyshell Error Messages later in this section.

ksh: ... If you receive an error message beginning with "ksh:", refer to the Keyshell Error Messages later in this section.

line not in TEKHEX. Issued by **mload**. In a data transfer, a line was encountered that was not in TEKHEX format.

line too long. A line was read that was too large for **dsc50**'s internal buffer.

load terminated. Issued by **mload**. The data transfer stopped prematurely.

mail saved in 'filename'. Your mail was saved in the specified filename. (This is an informational message, rather than an error.)

memory fault. Your program tried to access non-existent memory or memory outside of the program's fragment.

mismatch—password unchanged. Issued by **passwd**. The second time you typed your password was different from the first.

missing conjunction. You made an error in the **find** command syntax.

missing destination directory. Issued by **dsc50**. The destination directory does not exist or is a file.

missing flag argument. A required option was omitted from the command line.

missing number. An option was encountered which requires the next argument to be numeric, but the numeric argument was missing.

missing process ID argument. The **kill** command must specify a process ID.

missing string. A option was encountered which requires the next argument to be a string, but the string argument was missing.

missing string argument. The command requires a string argument.

missing time argument. The number of seconds to the **sleep** command needs to be specified.

more than one mutually exclusive flag chosen. At least two conflicting options were specified, or the same option appeared more than once in a command line.

mount device busy. An attempt was made to mount a device that was already mounted, or an attempt was made to unmount a device on which there is an active file (open file, current directory, mounted-on file, active text segment).

multiple rules lines for 'string'. Issued by **make**. More than one rule is applicable for updating the file.

name too long. Issued by **mv**, **dsc50**, **syschk**, and **fbr**. A pathname was encountered that was too large to fit in the program's internal buffer.

no children. Issued by the system call *wait*. The process executing the program containing the *wait* call either has no child processes, or a *wait* call has already been issued for all the process's child processes.

no more processes. Issued by the system call *fork*, while attempting to create a new process. Either the system's process table (an internal data structure) is full, or you may not create any more processes.

no namelist. Ps did not find the name list (symbol table) in the files it searched.

no room for columns. Issued by **pr**. There is not enough room on the paper for the number of columns you wish to print.

no shell. Issued by commands that execute a new shell, (such as **newgrp** and **su**) when the *exec* call, used to start the execution, fails.

no space. Issued by **shell.** Your shell has run out of memory. Try the command again when some of the current processes have terminated.

no space left on device. During a **write** to a data file, it was found that there is no free space on the device that contained the file.

No space left on volume. Issued by **dsc50**. There is not enough free space on your DOS/50 disk. Delete some files or use a fresh DOS/50 disk.

no such device. An attempt was made to apply an inappropriate system call to a device (such as reading a write-only device).

no such device or address. I/O was attempted to a special file that does not exist or has an invalid device designation.

No such DOS/50 file or directory. Issued by **dsc50**. The specified file does not exist on the DOS/50 disk.

no such file or directory. A directory or a file in a pathname does not exist.

no such group. Issued by **newgrp**. You may have misspelled the group name you are trying to refer to, or the group may not exist.

no such process. The process whose ID number was given to the *signal* or *ptrace* system call does not exist, or is already terminated.

no such tty. Issued by **write**. The */dev/ttyN* special file you specified (where N is an integer) does not exist.

no suffix list. Make needs a list of suffixes to form its rules. Check your makefile. Refer to the *Maintaining Files* section in this manual.

no write access to 'filename'. You do not have write access to the file. Use the **ls –l** command to check the mode of the file you are trying to access. Enter **man chmod** for more information.

non-numeric argument. Issued by **expr**. An expression contained a non-numeric argument. Enter **man expr** for more information.

not a directory. A file other than a directory was specified where a directory is required; for example, a file in a pathname or a file as an argument to the system call *chdir*.

Not a DOS/50 directory. Issued by **dsc50**. A DOS/50 file was specified in a context where a DOS/50 directory is required.

Not a DOS/50 volume. Issued by **dsc50**. The disk to be used as your DOS/50 disk does not have a DOS/50 disk structure. Use the **–m** option to initialize the DOS/50 disk before writing.

not a tty. Issued by **tty**. The device you specified was not a terminal.

not a typewriter. The file specified in **stty** or **getty** is not a terminal or one of the other devices to which these commands apply.

not enough contiguous space on archive. The free space on the **fbr** disk is too fragmented to store the desired files. Recreate the disk or delete some files to free larger chunks of the **fbr** disk.

not enough core. During the system calls *exec* or *break*, a program asks for more memory than the system is able to supply. This is not a temporary condition; the maximum memory size is constant. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers (used by TNIX for memory partitioning).

not enough memory. Issued by **syschk** and **dsc50**. The program could not allocate as much memory as it requires for its internal buffers. The task you are asking the command to perform may be too large for it to handle. If this error persists, contact your Tektronix field service representative.

not found. Issued by **sh**. The command was not in any of the directories in your PATH shell variable. Refer to the *TNIX Operating System* section of this manual for more information on the PATH variable.

not on that tty. Issued by **write**. The person you were writing to is not logged in on the terminal that you specified.

not owner. Typically, this error indicates an attempt to modify a file in a way allowed only to its owner or superuser. This error is also returned when a user attempts to perform tasks that are restricted to the superuser.

null name. Issued by **mail**. You need to enter the user name of the person you are sending to.

offset is greater than file size. Issued by **od**. The starting offset to begin dumping the file is set beyond the end of the file.

only one flag allowed with value arg. One of the options entered on the command line must match with a corresponding string or number argument one to one. An attempt was made to group such an option with other options.

operand follows operand. Issued by **find**. Enter **man find** for more information on the correct syntax.

out of buffers getting block 'block number'. Issued by **syschk** and **dsc50**. The program's internal cache of block buffers is exhausted. The task you are asking the command to perform may be too large for it to handle. If this error persists, contact your Tektronix field service representative.

out of free archive directory entries. The pool of file slots on the **fbr** disk has been exhausted. Delete some files from the **fbr** disk to make room.

out of memory. Ls tried to allocate some memory, but the request failed.

out of memory extracting 'filename'. Issued by **fbr**. Too much space is required to store the names of all the files to operate on.

out of memory recording links to 'filename'. Issued by **fbr**. Too much space is required to store the names of all the files to operate on. Split the desired operation into several smaller **fbr** commands.

out of memory storing the name 'filename'. Issued by **fbr**. Too much space is required to store the names of all the files to operate on. Split the desired operation into several smaller **fbr** commands.

parameter not set. A parameter required by a shell command has not been set.

parsing error. Issued by **find**. Enter **man find** for more information on the correct syntax.

password unchanged. You made a mistake while trying to change your password. Try again.

permission denied. An attempt was made to access a file in a way forbidden by the protection system. See the *TNIX Operating System* section of this manual for information about the file protection system.

phase error. 'filename' changed size. The specified file's size increased or decreased between the time that **fbr** allocated space for it and the time that its data was to be copied.

phase error. Cannot access 'filename'. Fbr makes two passes at the files to be operated on. This error means that some aspect of a file changed between the two passes. This particular message indicates that the specified file was deleted or otherwise made inaccessible.

phase error. Cannot open 'filename'. See the error message *phase error. Cannot access 'filename'*.

please use a longer password. Issued by **passwd**. A longer password is required.

please use at least one non-numeric character. The **passwd** program requires at least one letter or special character in your password.

read error 'block no'. Issued by **df**. An read error occurred while reading the specified files. Try again.

read error in '..'. Issued by **pwd**. A read error occurred while determining the pathname to the current directory.

read error on 'filename'. Issued by **syschk** and **dsc50**. A read error occurred while reading the specified file.

read-only file system. An attempt was made to modify a file or directory on a device mounted as read-only.

Regular expression error. Issued by **expr** and **grep**. There is something wrong with the regular expression you typed. See the *TNIX Editor* section of this manual for information on regular expressions.

regular expression too long. Issued by **egrep**. A regular expression was longer than 350 lines.

Relblk out of range (disk corrupt). Issued by **dsc50**. The specified file contains blocks that are not within the range of valid blocks on the DOS/50 disk. Delete the DOS/50 file.

Remove 'filename' 'octal number' mode?. The **rm** command issues this message when the file was not removed because of the protection mode. Enter **man chmod** for more information. Type "y" to remove the file.

repairs not allowed on mounted file system 'filename'. Issued by **syschk**. The command version of **syschk** does not repair file systems (the **–m** option). Use the stand-alone **syschk**.

restricted. You attempted to perform an operation restricted to the superuser.

result too large. The value of a function cannot be represented within machine precision.

seek error on 'filename'. An *lseek* system call failed. The offset parameter may be out of bounds.

short read on 'filename'. Issued by **dsc50**, **syschk**, and **fbr**. Fewer bytes were read than were requested; this is an I/O error. Enter **man read** for more information on the system call.

short write on 'filename'. Issued by **dsc50**, **syschk**, and **fbr**. Fewer bytes were written than were requested: this is an I/O error. Enter **man write** for more information on the system call.

signal 16. Signal number 16 was received by the *signal* system call. If this message is issued, there was no process to catch this signal after it was sent. Check your source code.

sorry. Issued by **newgrp**, **su**, and **passwd**. You did not provide the necessary validation information to perform the desired task.

sorry, path names including '..' aren't allowed. Issued by **mv**. It is illegal to move a directory using '..' in one of the pathnames.

spare sector in use. Issued by **syschk**. Each track of the fixed disk has one extra sector, used to replace one bad block. This message appears when **syschk** attempts to replace a second bad block on a track.

starting offset is greater than ending point. Od was told to begin dumping after the point it was told to stop dumping.

stray pms accept interrupt. The stand-alone **syschk** encountered a disk controller interrupt when one was not expected. Contact your Tektronix field service representative.

stray pms complete interrupt. The stand-alone **syschk** encountered a disk controller interrupt when it did not expect one. Contact your Tektronix field service representative.

stray pms utility interrupt. The stand-alone **syschk** encountered a disk controller interrupt when it did not expect one. Contact your Tektronix field service representative.

syntax error. An error was encountered while a command (such as **egrep** or **expr**) was attempting to parse an expression.

target name too long. Issued by **mv**. The name of the file into which you are moving a file exceeds 100 characters.

Temp file disappeared!. Issued by **passwd**. The temp file that **passwd** uses somehow got removed.

temporary file busy—try again. Passwd uses a common temporary file when it is logging someone in. This message indicates that the file is being used.

terminated. The shell issues this informational message to confirm that one of your processes was terminated.

***** Termination code 'decimal number'.** Issued by **make**. The error code that was returned caused **make** to halt.

text busy. See the error message *text file busy*.

text file busy. Issued by **sh**. An attempt was made either to execute a shareable read-only program that is currently open for writing (or reading), or to open for writing a shareable read-only program that is being executed.

timeout opening his tty. Issued by **write**. The terminal may not be plugged in or turned on.

tnix: bn="block number" er="errnum {,errnum ...}". Issued by the 8560's PMS Controller. "Block number" is the block number on which the error occurred. "Errnum" is the error number, as defined in Table 11-1 at the end of this section.

tnix: err on dev "devnum". Issued by the 8560's PMS Controller. "Devnum" is the major/minor device number of the device on which the error occurred. See Table 11-2 at the end of this section for a cross-reference listing of devices and major/minor device numbers.

tnix: error 2 on HSI device "devnum". Issued by the 8560's IOP or PMS Controller. You tried to use a tty port as an HSI port. "Devnum" is the major/minor device number of the device on which the error occurred. See Table 11-2 at the end of this section for a cross-reference listing of devices and major/minor device numbers.

tnix: error 4 on hsi device "devnum". Issued by the 8560's IOP or PMS Controller. Too much data is being sent out an HSI port. "Devnum" is the major/minor device number of the device on which the error occurred. See Table 11-2 at the end of this section for a cross-reference listing of devices and major/minor device numbers.

tnix: error 5 on hsi device "devnum". Issued by the 8560's IOP or PMS Controller. Not enough data is being sent out an HSI port. "Devnum" is the major/minor device number of the device on which the error occurred. See Table 11-2 at the end of this section for a cross-reference listing of devices and major/minor device numbers.

tnix: error 6 on hsi device "devnum". Issued by the 8560's IOP or PMS Controller. A transfer of data over an HSI port was not successful after many attempts. "Devnum" is the major/minor device number of the device on which the error occurred. See Table 11-2 at the end of this section for a cross-reference listing of devices and major/minor device numbers.

tnix: panic: .... Several error messages begin with *tnix: panic:*. These error messages will sometimes be issued before a system crash. Enter **man crash** for explanation of specific messages.

too big. During the system calls *exec* or *break*, a program asks for more memory than the system is able to supply. This is not a temporary condition; the maximum core size is constant. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers (used by TNIX for memory partitioning).

too many '('s. Issued by **expr**. Unmatched parenthesis in the expression.

too many command lines for 'string'. Issued by **make**. 'String' occurs on two dependency lines in your makefile, each with an associated command-list. If you need to retain two separate command-lists, use **make**'s double-colon syntax. Refer to the *Maintaining Files* section of this manual.

too many copy files; 'filename' not copied. Issued by **lpr**. You attempted to print too many files at once.

Too many DOS/50 directory levels. The specified pathname is at a deeper level in your DOS/50 disk than **dsc50** can traverse. Try changing your current DOS/50 directory to a directory nearer to the file you are working on.

Too many DOS/50 links. Issued by **dsc50**. The specified operation would result in more links to a given DOS/50 file than can be represented.

too many files. Ls cannot handle more than 1024 files.

too many keys. Sort cannot sort more than 10 keys.

too many links. You attempted to make more than 32767 links to a file.

Too many open DOS/50 files. Issued by **dsc50**. Try specifying fewer DOS/50 files in the operation.

too many open files. The default limit is 20 open files per process.

trace/BPT trap. Issued by **sh**. An erroneous BPT (breakpoint) assembly language instruction was executed. Check your source code.

transfer aborted. Issued by **mload**. An abort line was encountered in the data.

tty not accessible. Issued by **mesg**. Your */dev/tty* file was not found. Enter **man stat** for more information.

unknown error. Issued by the *perror* standard subroutine. The error number parameter specified in the *perror* call does not correspond to any errors in the *perror* list.

unknown group 'string'. Issued by **chgrp**. The group specified is not listed in the system groupname file. Use the correct group name.

unknown operator . Issued by **test**. Check your **test** syntax.

unknown option 'character'. Issued by **mail.** You tried to use an option that does not exist.

volume too small after initializing. Issued by **dsc50**. The DOS/50 disk is physically too small to contain the desired number of file slots. Use the **–a** option during disk initialization (**–m**) to request a smaller number of file slots.

write error. A physical I/O error occurred during a *write* system call. This error may occur on a system call following the one that actually caused the error.

write error on 'filename'. Issued by **dsc50**, **syschk**, and **fbr**. A write error occurred while writing to the specified file. Try the command again.

write error on copy of 'filename'. A **write** error occurred while **lpr** was copying your print file. Try the command again.

KEYSHELL ERROR MESSAGES

The following error messages are issued only by the three programs that interact to provide the Keyshell interface: **keynotes**, **kpp**, and **ksh**.

keynotes: bad index value for 'number' in file 'name'. The file containing *Explain Key Labels* information contains an error. Ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been isntalled again.

keynotes: bad line count 'count' for index 'number' in file 'name'. The file containing *Explain Key Labels* information contains an error. Ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

keynotes: environment variable TERM is not defined. The environment variable TERM has not been set to the name of your terminal. The command file *.profile* should do this when you log in; it may be missing or damaged. Ask your system manager for help in examining and perhaps replacing your *.profile*.

keynotes: file 'name' corrupted. The file containing *Explain Key Labels* information has been damaged. Ask your system manager to check the integrity of the system disk. Once disk integrity has been verified, ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

keynotes: missing arguments. The **keynotes** program was invoked incorrectly by **ksh**. Reinstall TNIX. Contact your Tektronix service representative if the error persists after you have installed TNIX again.

keynotes: 'name' not a keynotes file.. There are two possibilities: 1) **ksh** was invoked incorrectly in your login command file *.profile*. Ask your system manager for help in replacing *.profile*. 2) The file containing **ksh** *Explain Key Labels* information was altered. Ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

keynotes: no termcap entry for 'term'. The terminal data base file */etc/termcap* lacks a description of your terminal. Ask your system manager to verify that the TERM environment variable is set properly by the login command file *.profile* when you log in. If it is, then this message means that the Keyshell interface is not available for your terminal.

keynotes: no information for 'name' in file 'name'. There is no *Explain Keys* information for the current set of keys.

keynotes: 'termname' screen too narrow. Your terminal is not suitable for displaying *Explain Key Labels* information.

kpp: ... Error messages beginning "kpp:" result from errors in building customized **ksh** session files. Usually, one of the necessary files is missing or contains erroneous information. If you receive such an error message, ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

ksh: can't access tty driver. Indicates that **ksh** could not access a TNIX tty (terminal) device driver. Reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

ksh: can't find 'main'. The session file used by **ksh** lacks a "main" block. Ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

ksh: can't open session file 'name'. The session file needed by **ksh** is nonexistent or inaccessible. Ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

ksh: can't run two background processes. The session file used by **ksh** contains erroneous instructions. Ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

ksh: control character in argument on line 'n' of session file 'name'. The session file used by **ksh** contains an error. Ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

ksh: environment variable TERM is not defined. Ksh has found that the environment variable TERM has not been set to the name of your terminal. The command file *.profile* should be doing this when you log in; it may be missing or damaged. Ask your system manager for help in examining and perhaps replacing your copy of *.profile*.

ksh: extraneous argument on line 'n' of session file 'name'. The session file used by **ksh** contains an error. Ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

ksh: internal buffer overflow. This message usually results from building a very long command line. Try to accomplish the task using several shorter commands.

ksh: interrupted. Ksh was interrupted by a *quit* signal. Type **ksh** to invoke **ksh** again.

ksh: illegal argument on line 'n' of session file 'name'. The session file used by **ksh** contains an error. Ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

ksh: illegal delimiter on line 'n' of session file 'name'. The session file used by **ksh** contains an error. Ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not corect the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

ksh: 'main' didn't close. The session file used by **ksh** contains an error. Ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

ksh: missing argument on line 'n' of session file 'name'. The session file used by **ksh** contains an error. Ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX hs been installed again.

ksh: no session file. The session file used by **ksh** is missing. Ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

ksh: no termcap for 'term'. The terminal data base file */etc/termcap* lacks a description of your terminal. Ask your system manager to verify that the TERM environment variable is set properly by the login command file *.profile* when you log in. If it is, then this message means that the Keyshell interface is not available for your terminal.

ksh: out of memory space. If this message appears when you first log in, contact your Tektronix service representative. If it appears after you have been using **ksh** for a while, it indicates that there is no more room to accumulate command history. You may continue, but new commands will not be saved in your history list. Use the *exit* key to leave **ksh**, then type **ksh** to invoke it again and empty your history list.

ksh: redefinition of 'symbol' on line 'n' of session file 'name'. The session file used by **ksh** contains an error. Ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

ksh: system error – can't fork. Not enough TNIX process slots were available for **ksh** to use. Try the same task again later.

ksh: 'term' is not an acceptable terminal. Your terminal is not suitable for use with **ksh**.

ksh: 'term' missing capability 'xx'. Indicates that your terminal lacks one of the capabilities needed to run **ksh**.

ksh: unexpected end-of-file on line 'n' of session file 'name'. The session file used by **ksh** ended prematurely. Ask your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

ksh: unknown action on line 'n' of session file 'name'. The session file used by **ksh** contains an error. As your system manager to rebuild your **ksh** files with the **setuser** command. If that does not correct the problem, ask the system manager to reinstall TNIX. Contact your Tektronix service representative if the error persists after TNIX has been installed again.

REFERENCE TABLES

Table 10-1 explains "errnum" in the *tnix: bn="block number" er="errnum {,errnum ...}"* shell error message. Table 10-2 explains the "devnum" in several of the shell error messages beginning with *tnix:*.

Table 10-1
PMS Controller Error Number Definitions

Error Number	Explanation
01	8560 address impossibly large
02	Odd 8560 address
03	Odd byte count
05	Invalid device number
10	Invalid command code
11	Utility command issued to hard disk
12	Align command issued to hard disk
21	Drive not ready
22	No track zero signal detected
23	Data overrun error
24	ID field error in the Cyclic Redundancy Checksum
25	Bad cylinder address in ID field
26	Wrong cylinder address encountered in ID field
27	Bad ID fields
30	Missing data field address mark
31	Missing ID field address mark
33	Attempt to access sector beyond end of track
34	Invalid flexible cylinder address
35	Direct memory access timeout on disk read
36	Write protected diskette
37	Direct memory access timeout on disk write
41	Invalid hard disk command
42	Invalid parameter byte
43	Drive not busy
44	Drive fault
45	Illegal head or cylinder address
46	Sector not found
47	Data error
51	Timeout error
52	Hard disk positioner error
53	Drive fault during write
54	Hard disk performed retry after read/write error
55	Direct memory access timeout during hard disk read
56	Error correcting code performed
57	Direct memory access timeout during hard disk write
124	Flexible disk block number too large
141	No spare sector on specified track
142	Spare sector command did not select hard disk
143	Hard disk access timeout
144	Hard disk block number too large

Table 10-2
Major/Minor Device Numbers

Device	Major/Minor	Device	Major/Minor
aux1	5/0	hsix6	1/22
aux2	5/1	hsix7	1/23
console	0/8	kmem	2/1
fd0	0/4	lp1	5/0
hd0	0/0	lp2	5/1
hsi0	1/0	mem	2/0
hsi1	1/1	null	2/2
hsi2	1/2	rfd0	3/4
hsi3	1/3	rhd0	3/0
hsi4	1/4	swap	0/0
hsi5	1/5	tty	4/0
hsi6	1/6	tty0	0/8
hsi7	1/7	tty1	0/9
hsix0	1/16	tty2	0/10
hsix1	1/17	tty3	0/11
hsix2	1/18	tty4	0/12
hsix3	1/19	tty5	0/13
hsix4	1/20	tty6	0/14
hsix5	1/21	tty7	0/15

Section 11

GLOSSARY

Ancestor. See *parent.*

Argument. A name or expression in a TNIX command line that supplies information to the command, command file, or program to be executed.

Background Execution. The process of executing a list of one or more commands concurrently—without waiting for these commands to finish executing—by placing an ampersand (&) character at the end of a command line.

Bad Block. A block of disk space that is no longer usable due to physical damage. A bad block is not allocated once it has been marked by the TNIX **syschk** command.

Blank Interpretation. The process of parsing a shell command line into its constituent arguments. Variable and command substitutions are performed before the command line arguments are separated into individual arguments. The shell variable IFS defines characters that the shell uses to separate command line arguments.

Block Special File. A software device driver that transfers data to/from an I/O device in units of blocks.

Block. (1) Any 512-byte segment of disk space, memory, data, or program code. (2) In Tekhex protocol, any message up to 255 bytes in length.

Category C. Optional software supplied by Tektronix, for which Tektronix makes no warranty, express or implied, that the software is suitable for a specific purpose or that it performs any specific function correctly. Category C software includes:

- Optional Text Processing Package
- Optional Native Programming Package
- Optional Auxiliary Utilities Package

Character Special File. A software device driver that transfers data character-by-character to or from an I/O device (such as a terminal).

Child. The process created when an executing process creates another concurrently executing process with the **fork** TNIX system call. (The originating process is called the *parent* process.)

COM Interface. A communications interface similar to TERM mode, but more restrictive. With the COM interface, communicates with the 8560 primarily for the purpose of transferring object code and files. The COM interface is most useful for communicating with the 8001 and 8002A (which do not support TERM mode). See the *Communication with 8540s and 8550s* section of this manual for a brief description of the COM interface.

Command File. A file that contains commands to be processed by the TNIX shell or by a system program such as the linker, the library generator, or the ACE editor.

Command Substitution. A feature of the TNIX shell that replaces one or more commands enclosed in accent grave characters (') with the output from those commands, supplying the output of the accent-grave-enclosed characters as a command line argument to a command.

Control Character. A character whose ASCII code is in the hexadecimal range 00 to 1F, or 7F. Some control characters are entered by pressing special keys, such as TAB, ESC, or RETURN. Others are entered by simultaneously pressing the CTRL key and some other key.

Cooked Device. A software device driver that allows the system to process transferred data.

Current Directory. The directory that you are currently accessing. The current directory is used as a starting place when a full pathname (a name that begins with "/") is not given. The current directory is sometimes referred to as the working directory.

Daemon. An independently executing process that asynchronously performs periodic system maintenance tasks, such as printing files from a spool directory. (A spool directory contains files that are queued, waiting to be printed on the line printer.)

Default. A predefined value for a command parameter, used when no value for the parameter is explicitly specified.

Descendant. File F is termed a descendant of directory D if:

- D contains F, or
- a descendant of D contains F.

Device. (1) A piece of computer hardware. (2) An instrument attached to the 8560 that is used for data entry, storage, and display. For device I/O, TNIX treats devices as special files. Thus, you can direct a program's output to a special file the same way you would direct that program's output to a regular file. All special files are located in the */dev* directory. See *logical device.*

Device Number. See *major device number.*

Directory. A file that contains only pointers (called links) to other files. A file that is pointed to by a directory is said to reside in the directory; every file resides in at least one directory. Similarly, a directory is said to contain each file that the directory points to. The *mkdir* command creates a new directory.

DOS/50. The operating system of the 8550 Microcomputer Development System.

Download. To transfer data from a host such as the 8560 to a microcomputer development system (such as the 8540).

Environment. A set of string-valued parameters maintained by the shell. These parameters can be accessed by programs executed by the shell. The environment consists of:

- the current directory,
- your user name,
- your group name,
- the default shell variables HOME, IFS, PATH, PS1, PS2, and
- any shell variables that you have defined and "exported" with the shell's **export** command.

Execute Permission. The ability to execute a file or search a directory. Permission is regulated by mode bits, which can be set with the **chmod** command.

File. The fundamental unit of data storage used by the TNIX operating system. A file is a collection of logically related information that is stored on a backup tape, a flexible disk, or a fixed disk drive. TNIX does not assign any specific structure to a file.

File Descriptor. An integer in the range 0—20 that is returned by a **creat**, **dup**, **open**, or **pipe** TNIX system call. The file descriptor, instead of a filename, is used as a parameter to TNIX system calls that access open files. These TNIX system calls include **close**, **ioctl**, **lseek**, **read**, **fstat**, and **write**. The file descriptors 0, 1, and 2 are assigned to standard input, standard output, and standard error, respectively.

File Tree. A directory and all its descendants.

Filename. A sequence of 1 to 14 characters that specifies a file. The "/" and the null character *cannot* be used in a filename. You should also avoid using control characters, blank, newline, tab, or the following special characters in filenames:

& ' ' " ; : ? ! - $ * < > [ ] ( ) \

Filesystem. A complete directory hierarchy that may extend over one or more fixed disk drives. The "root" filesystem is the "/" filesystem—other filesystems are *mounted* on directories within the root filesystem. Each filesystem has a maximum number of files that may be created on it. Files may not be linked across filesystems.

Filter. A program such as **grep** or **tr** that accepts data from standard input, performs some transformation upon the data, and writes the data to standard output.

Fork. A system call that creates a new (child) process that executes concurrently with the calling, or parent, process. The child process inherits the open files of the parent and executes concurrently with the parent. To avoid concurrent execution of the child and parent process, the parent process can execute the **wait** TNIX system call, which causes it to pause until the child process terminates.

Group. Groups provide a simple method for restricting read/write/execute privileges for specific collections of files and directories to a subset of the users on a specific machine. Each group is assigned a specific entry in the */etc/group* file; users who have access to that group are listed in that group's entry in the */etc/group* file.

Group ID. The numerical identification of a user as a member of a group of users working on related information.

Groupname. The alphanumeric name of a group.

Host. A computer system (such as the 8560) that is used to prepare and maintain programs that are tested and debugged on a workstation such as the 8540.

I-node. An entry in a TNIX data structure that describes:

- the physical location of a file on a fixed disk,
- the ownership and protection modes of a file,
- the last access date of a file, and
- the number of filenames associated with a file.

I/O Redirection. The ability of the TNIX shell to take the source and destination of a program's input and output from files other than standard input and standard output.

K. 1024 bytes (400 hexadecimal).

Kernel. The main memory-resident part of the TNIX operating system.

Keyshell. An interface to TNIX that enables you to enter commands by pressing function keys as well as typing the commands literally.

Library. A collection of object modules that usually contains commonly used subroutines.

Link. (1) A pointer from a directory to a file. The **ln** command allows you to create multiple pointers or *links* to a single file. (2) To merge object modules into a load module using the linker.

Linker. The system program that combines object modules into a single executable load module. The Tektronix Linker is described in the *8500 Modular MDL Series Assembler Core Users Manual for B Series Assemblers.*

Logical Device. See *special file.*

Log In. To sign into the TNIX operating system.

Login Name. The name by which the system recognizes a particular user at login. A login name is typically 1 to 8 lowercase letters. The login name is also referred to as the username.

Major Device Number. An entry in a TNIX data structure that specifies the type of software device driver used to perform a data transfer operation. Each type of I/O device (i.e., a fixed disk drive or a terminal) uses the same software device driver, and therefore has the same major device number. Individual devices of each type are specified by the minor device number.

Make. Described in Section 6 of this manual, the **make** program can be used to perform the clerical tasks associated with updating and maintaining interrelated program modules. With moderately complex programs, this can be an almost indispensable time-saver and program bug-saver.

Man. A TNIX command that provides access to online information. For example, **man cd** displays information about the **cd** command.

Metacharacter. A character used to match specific patterns of characters. Metacharacters are used by the TNIX shell, **ed**, **sed**, **grep**, **awk**, and a number of other programs. For example, the TNIX shell substitutes any filenames that match the pattern of characters represented by the series of one or more metacharacters. Regular expressions (described later in this *Glossary*) are defined by a series of metacharacters.

Minor Device Number. An entry in a TNIX data structure that specifies the physical device to be accessed for a data transfer operation. For example, if there are two line printers connected to your system, TNIX would use the major device number to select the software device driver for the printer, and the minor device number (internally) to specify which printer to transfer data to.

Mode Bits. The 12 protection bits associated with a file. The first 9 bits specify read, write, and execute access for the owner, members of the owner's group, and remaining users. The last 3 bits specify that the program set the group/user ID on execution, and that the program remains in main memory for a minimum amount of time (use this mode for heavily used programs, such as **ls**). Use the **chmod** command to change a file's mode bits, and the shell's **umask** command to set the default file creation mode bits.

Mount. To associate a physical device (such as an 8503 Disk Expansion Unit) with a directory in the file tree. The data stored on the device is then accessible as subcomponents of the tree. See your *8560 Series System Manager's Guide* for more information.

Multiplexed Special File. A special file that allows multiple communication paths between one or more executing processes.

Newline. The newline character is the ASCII LF (octal 12) character, and is used to separate lines in TNIX files. You can enter a newline character into a file by pressing the LINEFEED or RETURN key on your terminal.

OS/40. The operating system of the 8540 Integration Unit.

Parameter. A name or expression in a TNIX command line that supplies information to the command, command file, or program to be executed.

Parent. (1) The directory that contains a specific directory or file. For example, */usr/gandalf* is the parent directory of */usr/gandalf/projects*. (2) The originating process when an executing process creates another concurrently executing process with the TNIX system call **fork**. (The newly created process is called the *child* process.)

Password. The private sequence of characters that allows you to be identified to the system by your username. Each username may have an associated password, initially assigned by the system manager. You can change your password with the **passwd** command. Passwords are stored in an encrypted format in the */etc/password* file.

Pathname. A sequence of directory names separated by slashes and ending in a filename, that defines a path to a file. For example, */usr/tektronix/myprog* is a pathname for the file *myprog* in the directory *tektronix*. If */usr/tektronix* is the current directory, *myprog* is a synonym for */usr/tektronix/myprog*.

Permission. See *execute permission*, *read permission*, and *write permission*, in this *Glossary*.

Pipe. A communication channel used to transfer data between two executing programs. Pipes may be established from the shell using the "|" character.

Pipeline. A sequence of one or more commands separated by "|". The standard output of each command except the last is connected by a pipe to the standard input of the next command.

Process. An independently executing program with its own current directory, open files, user ID, group ID, and other process-specific information. Each process has its own memory for stacks, variables, and program information.

Process ID. A number in the range 1—30000 that uniquely identifies an executing process to TNIX.

Profile. (1) A file named *.profile* in your HOME directory that contains a list of commands executed by the TNIX shell when you first log into TNIX. (2) To obtain an execution trace of a program as an aid in debugging the program.

Prompt. The character that a program displays to tell you that it is waiting for input. The TNIX shell has three default prompt characters: "$", ">", and "#".

Queue. (1) To place files to be printed or processes to be executed into a waiting line. (2) The waiting line itself. Files are said to be "spooled" or "queued" when they are in a print queue.

Raw Device. A software device driver that transfers data without any processing by the system.

Read Permission. The ability to read from a file or list the contents of a directory. Permission is regulated by mode bits, and can be set with the **chmod** command.

Regular Expression. A series of one or more characters used to match specific patterns of characters. Regular expressions are used by the TNIX shell, **ed**, **sed**, **grep**, **awk**, and a number of other programs. Regular expression syntax varies from program to program. Be sure to consult the documentation for regular expression syntax for a specific program.

Regular File. Any file that is not a directory or a special file.

Root Directory. The directory (represented by "/") at the top (root) of the TNIX file tree. Except for the root directory itself, all files are descendants of the root directory.

Shell. A command interpreter that serves as an interface between you and the TNIX operating system. (See the *TNIX Operating System* section of this manual.)

Shell Procedure. A shell program. The *Shell Programming* section of this manual describes the shell programming language.

Shell Variable. Any command argument in a shell command line whose first character is a "$". Before executing the command specified in the command line, the TNIX shell replaces the specified shell variable with that variable's value.

Slave Computer. A microcomputer development system (such as the 8540) that is connected to a host computer (such as the 8560).

Special File. An entry in a directory, usually the */dev* directory. Each special file is associated with a software device driver, and read, write, open, and close operations on a special file are actually performed by the associated software device. The TNIX operating system handles I/O to devices as if they were part of the file system. (See *block special file*, *character special file*, and *multiplexed special file* in this *Glossary*.)

Standard Error. A file descriptor (2) that is used by the TNIX shell to display error messages or other information that should not be intermingled with standard output. Normally sent to your terminal, the standard error output can be redirected to other files or devices.

Standard Input. A file descriptor (0) that is used by the TNIX shell to supply input to a program. Standard input is usually your terminal, but may be redefined with the <, <<, or ¦ characters.

Standard Output. A file descriptor (1) that is used by the TNIX shell to display output from a program. Standard output is usually your system terminal, but may be redefined with the >, >>, or ¦ characters.

String. A series of one or more ASCII characters.

Subdevice Number. See *minor device number.*

Subshell. A shell process created by a parent shell process. A subshell inherits the parent shell's environment, which includes the current directory, group ID, and variable settings, but may redefine its environment.

Superuser. Any user logged into the "root" account, usually to perform some system maintenance operation (such as file system backup, restore, or repair). See the *8560 Series System Manager's Guide* for further information.

NOTE

Normal file protection does not apply to the superuser. Some system calls are also restricted for use only by the superuser.

System Call. An operating system service routine executed directly by the TNIX operating system. System calls are entered directly via software interrupts (traps) as rather than by the typical subroutine calling sequence.

System Manager. The person responsible for system operations such as system backups and restores, and software installation. The system manager usually logs into the "root" account in order to perform these operations. See the *8560 Series System Manager's Guide* for further information.

Tekhex. Tektronix Hexadecimal Format: a format for representing the contents of a block of memory as an ASCII sequence of hexadecimal digits. An extended version of Tekhex supports 16-bit microprocessors and symbolic debug. (Refer to the *8540* or *8550 System Users Manual.*)

TERM Mode. A mode of communication between an 8560 and an 8540 (or 8550), in which workstation commands can be entered just as if they were TNIX commands. The 8560 recognizes the workstation commands and passes them to the 8540 (or 8550). See the *Communication with 8540s and 8550s* section of this manual for a detailed description of the TERM mode interface.

TNIX. The operating system of the 8560 Series Multi-User Software Development Unit.

Tree. A data structure. The TNIX filesystem structure is referred to as a file tree, because the data structure used by the TNIX filesystem is an inverted tree structure.

Unmount. To dissociate a physical device from a position in a file tree. The data stored on the device can no longer be accessed as subcomponents of the tree.

Unlink. To delete a pointer to a file from a directory. A file is destroyed when its last link is removed.

Upload. To transfer data from a workstation such as the 8540 or 8550 to a host, in this case, the 8560.

Username. The name by which the system recognizes a particular user at login. A username is typically 1 to 8 lowercase letters. The username is also referred to as the login name.

Wildcard Character. See *metacharacter.*

Word. A sequence of non-blank characters.

Working Directory. The directory that you are currently accessing. The working directory is used as a starting place when a full pathname (a name that begins with "/") is not given. The working directory is sometimes referred to as the current directory.

Write Permission. The ability to write to a file or to create and delete files in a directory. Permission is regulated by mode bits, and can be set with the **chmod** command.

Section 12

INDEX

C

D

E

F

G

H

I

J-K

L

M

N

O

P

Q-R

S

MANUAL CHANGE INFORMATION

At Tektronix, we continually strive to keep up with latest electronic developments by adding circuit and component improvements to our instruments as soon as they are developed and tested.

Sometimes, due to printing and shipping requirements, we can't get these changes immediately into printed manuals. Hence, your manual may contain new change information on following pages.

A single change may affect several sections. Since the change information sheets are carried in the manual until all changes are permanently entered, some duplication may occur. If no such change pages appear following this page, your manual is correct as printed.