EY-1034E-SG-0002

VAX/VMS Users Introduction

Student Guide

Prepared by Educational Services
of
Digital Equipment Corporation

Printed in U.S.A.

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

The software described in this document is furnished under a license and may not be used or copied except in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by Digital.

TABLE OF CONTENTS

Chapter Page

Table of Contents (cont.)

Chapter Page

INTRODUCTION

The purpose of this document is to introduce you to VAX/VMS. The document is divided into chapters, where each chapter discusses a different aspect of VAX/VMS from the user's point of view.

Chapter 1 provides an overview of the user's environment, discussing the software and hardware available with VAX/VMS.

Chapter 2 gets you started, by discussing how to log in and out, use the terminal, enter commands, get help when needed, and obtain information about or modify the user environment.

Chapter 3 discusses file naming conventions, directory structure, use of defaults, and deciphering error messages.

Chapter 4 discusses file creation using the EDT editor, and file manipulation commands.

Chapter 5 discusses program development in general, including program examples for several languages supported on VAX/VMS (MACRO, FORTRAN, COBOL, BASIC, PASCAL). The VAX-11 Symbolic Debugger is also discussed in this chapter.

Chapter 6 introduces command procedures and symbols, methods that can be used to simplify a user session.

Chapter 7 provides an overview of the RUNOFF text formatter, including examples and a summary of popular commands.

Chapter 8 discusses some other useful utilities, MAIL and PHONE.

CHAPTER 1

THE USER ENVIRONMENT

1.Ø THE USER ENVIRONMENT

A computer system consists of two major parts:

- o Hardware
- o Software

Hardware is a term used to refer to the physical computer, which is manufactured in a factory.

Software is a term used to refer to the programs that contain instructions to be performed by the hardware.

The combination of hardware and software forms a system. Many types of hardware and software exist, so computer systems do not have to be, and rarely are, identical.

A user's environment is defined by the combination of hardware and software on his/her particular system. Since the elements forming each system may not be the same, a user of one system will probably work in a different environment than a user on another system.

Each system is managed by a system manager. The system manager is familiar with the system environment, and can further restrict each user's environment.

1.1 THE HARDWARE

The hardware on a system is generally divided into three parts - the central processing unit (CPU), main memory, and peripheral devices.

The central processing unit is where most of the work is done on a computer system. In the VAX family of computers, there are four models of the CPU, including the 11-78Ø, 11-75Ø, and 11-73Ø. The 11-78Ø model is larger than the 11-75Ø. The 11-73Ø is the smallest model. All do the same job; some faster than others. There is usually only one CPU per system. The 11-782 (larger than the 11-78Ø) uses two CPU's, one as the primary worker, and the other as the secondary worker. Work is shared between the two processors according to rules set up by the designers.

Main memory is used for temporary storage of instructions and data. Main memory can be installed in units, so the amount of memory on a system can vary. Battery backup is available so the contents of main memory are not lost in the case of a power failure. The system manager can set up the system to start automatically after a failure (such as a power outage), and restore the contents of memory. Therefore, with battery backup, work is rarely lost.

Peripheral devices include disk drives, magnetic tape units, printers, terminals, and card readers.

Each disk or magnetic tape is referred to as a volume in this document. The term **device** is used to refer to the physical equipment where the volume is mounted.

Disks are used by the system to store currently used information. A disk can be placed in a disk drive or stored in a cabinet in the same way a record can be played on a record player or stored in a cabinet. Although several disk drives may be attached to a system, the user's information is normally recorded on one disk only, which may be mounted in any drive.

In the same way, a person may own several record players to play records on. If a particular song is recorded on one record only, the person may play the record on any of the players and hear the same song. If the creators of another record decided to include the same song, or a variation of the song, on their record, the song would be on more than one record. In the same way, the same information, or different versions of the information, may be stored on more than one disk.

Magnetic tapes are normally used to store information not in current use, to free up storage space on the disks. The owner of the disk decides what will be stored on tape and/or removed from the disk.

Many different types of disk and magnetic tape drives can be installed as part of a VAX/VMS system. The storage of information on disks and magnetic tapes is handled by the system and the system manager. This document assumes the user will not be handling disks or magnetic tapes.

Most users of VAX/VMS work with printers and terminals.

Several types of printers are available. The system manager chooses one of the printers on the system to be the default printer. All files to be printed are sent to the default printer unless the user specifies otherwise.

Several types of terminals are available. Some have a video screen, such as the VT52 and VT1ØØ. Others are hardcopy terminals using paper, such as the LA36 or LA12Ø (see Figure 1-1). A standard keyboard is built into all DIGITAL terminals (see Figure 1-2). This document assumes DIGITAL terminals are being used.

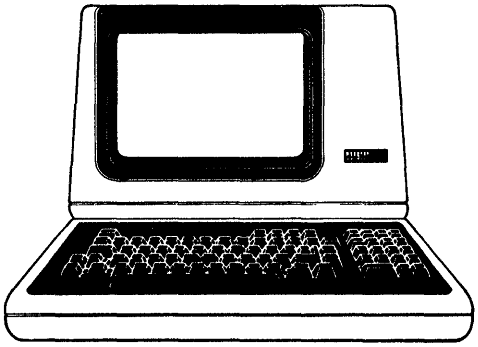


Figure 1-1

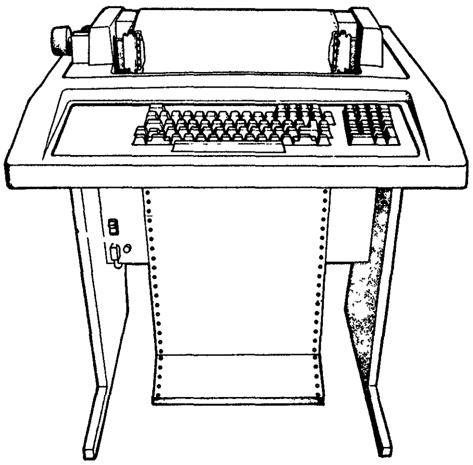


Figure 1-2

1.2 THE SOFTWARE

The software on a system is generally divided into two major parts - application software and system software.

Application software includes programs written by users of the system for specific purposes, such as budgeting, processing the payroll, running machines, or keeping personnel records up-to-date.

System software includes programs written by the creators of the system for such purposes as coordinating users, sharing resources, running the hardware, and helping the user communicate with the system.

1.3 RESTRICTING THE USER'S ENVIRONMENT

A user can be restricted from access to:

- The system (i.e., not allowed to work on the system)
- Other users (i.e., so can not affect the work of other users)
- Certain kinds of software (such as system programs)
- Particular kinds of hardware

Information about each user is stored in a special file, called the User Authorization File (UAF), on the system. The system manager can modify any of information stored there to allow the user more access to hardware and software, or to restrict the user further.

The information in the UAF includes:

- The user's name and password - needed for access to the system
- Privileges - to allow or disallow access to hardware and/or software
- Limits - to restrict the use of system resources
- UIC - User Identification Code
- Priority - used by the scheduler to coordinate users - on a 'higher priority - first serve' basis

When a user logs in, VMS uses this information to create a **process**. A process contains a complete description of the user's environment, including all of the information from the UAF, what the user is doing, and what part of memory the user is working in. Therefore, the process is equivalent to the user's environment. Each user works in the context of their own process. VMS coordinates, manages, and allocates resources to processes, not users.

Processes are created for the purpose of running programs. When a user logs in, a special kind of process is created - an **interactive** process. The term interactive means that the user is interacting directly with the system, usually via a terminal.

VMS runs a program for interactive processes as soon as they are created. The default program may be changed by the system manager, but this document assumes that the program is the command language interpreter for the DIGITAL Command Language (DCL).

The DCL interpreter accepts a DCL command input by the user and runs the system program corresponding to that command. One DCL command is the RUN command, which can be used to execute user programs. After user or system programs have completed, VMS runs the DCL interpreter again, so the process will not be deleted. (If a program is not executing in a process, VMS deletes the process.)

The user will know if the DCL interpreter program is executing by the presence of the DCL prompt, $ (a dollar sign). The dollar sign prompt indicates that the DCL interpreter is ready to receive a command from the user. If the dollar sign prompt is not present, another program is probably executing, and DCL commands should not be input.

Interactive processes are deleted by VMS when the user logs off the system. Resources which were used by that process are then available for use by other processes.

CHAPTER 2

GETTING STARTED

2.1 LOGGING IN AND OUT

Before you can log into the system you must obtain permission to use the computer. The system manager is usually the person to contact. The system manager will give you a username and password that will permit you to use the facilities of the system.

Once you have a username and password you can log in. To log in to the VAX/VMS system, do one of the following:

- Press the <RETURN> key on the right side of the keyboard
- Press the control key <CTRL> on the left side of the keyboard. Hold it down and press the C or Y key (both achieve the same results).

You should see a request for your user name in the format:

```
        Username:
```

If you do not see the prompt:

- First, check to see if the terminal is plugged in and turned on.
- Then, try again.
- If you still do not see the prompt, get help from your system manager or designated expert.

If you received the prompt, enter your user name. The system should output another prompt requesting your password in the form:

```
Password:
```

Enter your password. The password does not echo (i.e., you can not see what you type), so type carefully.

The system should output a welcome message. Some systems also output site-specific informational messages. (These informational messages can be changed, and added to, by the system manager.)

If the system outputs an error message instead of a welcome message:

- Start over and enter the information more carefully

- If you still receive an error message, notify your system manager or designated expert. (Sometimes the information recorded in the UAF corresponding to your user name is not correct. Sometimes the information has not been recorded. By notifying the system manager, the problem should be corrected so you will not receive any more error messages.)

If the system outputs an informational message such as 'system busy - try again later', then obey the message.

Assuming you have been successful in logging in, you should see the dollar sign prompt, $, at the left side of your terminal screen. The $ was output by the DCL interpreter program executing in your process. The DCL interpreter is ready to receive a valid DCL command.

One valid DCL command is LOGOUT. If you enter this command, your process is deleted and its resources are returned to the system.

```
$LOGOUT
```

The examples that follow show both a successful and unsuccessful attempt to login to the system.

Example 1 -- Successful Login

```
<CR>
Username:SMITH
Password:

Welcome to VMS V3.Ø

$
```

Example 2 -- Unsuccessful Login

```
<CR>
Username:SMITH
Password:
User Authorization Failure
```

2.2 SPECIAL TERMINAL KEYS

A diagram of the standard DIGITAL keyboard can be seen in chapter 1, Figure 1-2. The following terminal keys can be used while you are logged in to correct errors or modify the behavior of programs:

- o DELETE - Used to delete the character just entered on the terminal

  For example, If you enter PAPEF when you meant to enter PAPER, press the DELETE key after entering the F.

  On a video screen, the F will be erased, leaving the cursor after the E. You can then enter the correct letter, R.

  When working on a hardcopy terminal, the deleted character will be echoed, preceded by a backslash character. When the correct letter is entered, another backslash character will appear on the paper, followed by the new letter.

  ```
  PAPEF/F/R
  ```

- o BACKSPACE - Do not use! The character entered by this key is unacceptable input to the DCL interpreter or a compiler.

- CTRL - This key is to be used in conjunction with one of the following keys by holding it down while pressing one of them:

  - C or Y - suspends the current command line or currently executing program. The dollar sign prompt is then output.

  - R - retypes the current input line on the terminal. CTRL-R is useful on hardcopy terminals after several corrections have been made to an input line.

    ```
    Papef/f/r is a uf/f/seful tb/b/ool   (user types CTRL-R)

    Paper is a useful tool           (line is retyped as the
                                      computer will see it.
                                      Input may continue at
                                      the end of the line.)
    ```

  - U - cancels the current command line

  - S - stops the display of information on the terminal screen

  - Q - continues printing output stopped with the CTRL/S on the terminal screen

  - O - suppresses output to the terminal screen but allows program to continue. Entering another CTRL-O reverses the effect so the output can be seen again. (The information output by the program while output to the terminal screen is suppressed is never seen by the user.)

NOTE: Sometimes a terminal will not respond to a user, and appears to have stopped working. Often, this is because the user accidentally entered a CTRL-S or a CTRL-O. The terminal will usually respond if a CTRL-Q or CTRL-O is entered. If that fails, enter a CTRL-Y.

2.3 DCL COMMAND FORMAT

Any valid DCL command can be input by the user when the $ prompt is seen. The general format of all DCL commands is the same. However, some commands may be more explicitly defined or modified through the use of command options, parameters and qualifiers.

Table 2-1 lists the major command formats and examples of commands using those formats.

Table 2-1

Command Format	Example
$command	$LOGOUT
$command option	$SHOW SYSTEM
$command option/qualifier	$SHOW DEVICE/ALL
$command parameter	$TYPE FILE.DAT
$command/qualifier parameter	$DIRECTORY/FULL FILE.DAT
$command parameter/qualifier	$PRINT FILE.DAT/COPIES=2
$command parameter,parameter	$PRINT FILE.DAT,TEST.FOR
$command param,param/qualif	$PRINT A.DAT,B.FOR/COPIES=4

The first four characters of any DCL command, option, or qualifier uniquely identifies it to the DCL interpreter. For example, PRINT can be shortened to PRIN, and DIRECTORY can be shortened to DIRE. Many commands are uniquely defined by fewer characters than four, so the user rarely needs to enter the entire command. For example, DIRECTORY can actually be shortened to DIR.

Many commands require an option or parameter so the DCL interpreter will know exactly what to do. The interpreter will prompt the user for missing information. For example, the PRINT command prompts for a file name.

```
$PRINT                   (user pressed <RETURN> )
$_file:                  (system prompt...user
                          should input file name)
```

As soon as the DCL interpreter has received all required information, it will invoke the corresponding system program. For example, the PRINT command requires only one file name. If a user enters one file name and presses the carriage return, the file will be printed. If the user intends to enter more than one file name, the carriage return should not be pressed until all file names have been entered. For example:

```
$PRINT                     (user pressed <RETURN>)
$_file: FILE.DAT           (user enters file name and presses
                            <RETURN>.  File is printed)


$PRINT
$_file: FILE.DAT,A.DAT,B.DAT    (user list names and does not
                                 press <RETURN> until all have
                                 been listed.  All files are
                                 printed.)
```

If a user needs to print so many files that the end of the line is reached before all files have been listed, a continuation marker can be placed at the end of the line. The continuation marker accepted by the DCL interpreter is - (a hyphen). The user can press the carriage return after entering the hyphen, and continue to input file names after the $_ prompt on the next line. A carriage return pressed after the last name causes all listed files to be printed. The continuation marker can be used with any DCL command. For example:

```
$PRINT FILE.DAT,A.DAT,B.DAT,-
$_TEST.FOR,PAYROLL.DAT
```

2.4 GETTING HELP

All commands listed in Table 2-1 are valid DCL commands. More information is available on-line for every DCL command. To obtain this information, enter the command HELP when the $ prompt is seen.

An alphabetical listing of all DCL commands and other selected topics will be seen. The HELP program then prompts for a topic. The name of any topic listed can be input after the prompt. Information about the topic will be output, including a statement "additional information available" preceding a list of subtopics, and a prompt for a subtopic.

Information about a subtopic listed can be obtained by inputting its name. If a carriage return is entered instead, the topic prompt will be output. If another carriage return is entered, the user will see the $ prompt. For example:

```
 $HELP
        (Alphabetical list of commands and topics)


 Topic? PRINT               (user enters name of topic)

        (general information about topic)
        (subtopics listed if available)


 Subtopic? /COPIES          (user enters name of subtopic)

         (information about subtopic is output)

 Subtopic?                     (user presses <RETURN> )

 Topic?                        (user presses <RETURN> )

$
```

NOTES:

1. The three words: options, parameters, and/or qualifiers are usually included in the list of subtopics for commands. Any of these may be entered as a subtopic to obtain general information. For example:

   ```
   Subtopic? parameters
   ```

2. If the subtopic is a command qualifier, the / is part of the name of the qualifier, as seen with /COPIES.

3. Another way to exit from the HELP program is by inputting a CTRL-C or CTRL-Y.

4. The HELP command accepts a topic and/or subtopic as part of the HELP command to obtain information more quickly. For example:

   ```
   $HELP topic subtopic
   ```

   Some examples of this include:

   ```
   $HELP SHOW SYSTEM
   $HELP DIRECTORY
   $HELP PRINT/COPIES
   ```

2.5 OBTAINING INFORMATION ABOUT THE ENVIRONMENT

The environment of a user is defined by the hardware on the system, the software available, and the information recorded about the user in the UAF.

Users can look at their environment through the use of one or more DCL commands listed in Table 2-2. Use HELP to find out more information about these commands.

Table 2-2 Commands to obtain information about environment

Information desired	Command to use
List of all processes on system	$SHOW SYSTEM
Information about own process	$SHOW PROCESS/ALL
Current statistics on own process	$SHOW STATUS
*Current position (device and directory)	$SHOW DEFAULT
Current system date and time	$SHOW TIME
Characteristics of own terminal	$SHOW TERMINAL
Characteristics of other devices	$SHOW DEVICE

*discussed in Chapter 3 of this document.

2.6 MODIFYING THE ENVIRONMENT

Users can change some of the characteristics of their environment. Table 2-3 lists the commands used to change typically modified characteristics. Use HELP to obtain more information about these commands.

Table 2-3 Commands used to modify user environment

Characteristic	Command
Password	$SET PASSWORD
Width of line on terminal	$SET TERMINAL/WIDTH=132 $SET TERMINAL/WIDTH=8Ø
*Default position (device and directory)	$SET DEFAULT [directory-name]

*discussed in Chapter 3 of this document

CHAPTER 3

FILE NAMING AND MANIPULATING

3.1 FILE CONCEPTS

The following analogy should help you understand how information is stored and accessed on VAX/VMS.

A large company, called WERGRATE, owns a building. The building is divided into many rooms. Some of these rooms are set aside for the storage of information. Filing cabinets line the walls of each of these storage rooms. File folders containing information are stored in most of the cabinets.

In this analogy, we have defined several places:

The building

Rooms in the building

Filing cabinets in each room

One or more file folders in the cabinet

Many different types of information can be stored in the file folders, such as drawings, reports, and personnel records.

Many different kinds of files can be stored on a computer system. A file stored on a computer system can contain such things as text, source code, object code, or executable code. Files are created by an editor, a compiler, the linker, or other utilities. Normally, a file is stored on a disk or magnetic tape.

The storage areas in a company correspond to storage areas in a computer system as seen in Table 3-1.

Table 3-1 Correspondence between a company and a VAX system

A company	A VAX system
The building	A node
A room	A device
A filing cabinet	A directory
A file folder	A file

To send a person to retrieve a certain file folder, directions to the folder must be specified. The person must know which building to enter, where the correct room is, and which filing cabinet to open to access the folder. It is assumed the person sent is familiar with buildings, rooms, file cabinets, and folders. However, if the person is given incorrect directions, the folder may not be found, or a different folder may be retrieved.

To send the computer system to access a file, directions to the file, called a file specification, must be given to the system. In VAX/VMS, a file specification includes the names of the node, device, directory, and file. The system is familiar with nodes, devices, directories, and file names, and will attempt to locate the file as specified. If the user gives the system an incorrect file specification, the system may respond with an error message, or by retrieving a different file than the user intended.

3.2 SPECIFYING FILES

A file specification has the following format:

```
Node::Device:[Directory]File_name.File_type;Version_number
```

The fields of a file specification are discussed below.

- Node:: - the name of the system connected to the device where the file resides.
- Device: - the name of the device containing the volume (disk pack, magnetic tape) where the file is stored. Several devices may be connected to the user's system. Volumes can be moved from device to device. The information stored on a volume can be accessed only by specifying the name of the device where the volume is currently mounted. The system will respond with an error message if the volume is not available.
- [Directory] - the name of a special file, a directory file, where the name of the file is listed. The directory file is stored on the same volume as the file. Directory files are discussed further in section 3.2.2.
- File_name - any name chosen by the user. The name usually corresponds to the contents of the file.
- .File_type - should indicate the kind of information stored in the file, such as text (.TXT), data (.DAT), FORTRAN source code (.FOR), object code (.OBJ). The file type may also be chosen by the user, and does not have to correspond to the contents of the file.
- ;Version_number - indicates whether this is the first, second, third, etc. version of the file. When a file is created, the system assigns it a version number of 1. If the file is modified, the modified version is assigned the version number of 2. Each new modification is assigned a new number (increment is 1).

For example:

If the node is NODEA ,

the device is DRA3 ,

the directory is WHITE ,

the filename is MYFILE ,

the filetype is TXT ,

and the version number is 4 ,

then the full VMS file specification is:

```
NODEA::DRA3:[WHITE]MYFILE.TXT;4
```

The following are other examples of complete file specifications:

```
ENGNDE::DRAØ:[BROWN]TESTFIL.DAT;2

DEPTØ1::DBB3:[SERGIO]DRAWING4.TXT;33

ACCTNG::DBA1:[MANAGER]BUDGET.FOR;1

ACCTNG::DRAØ:[SYSEXE]HELP.EXE;1
```

3.2.1 FILE SPECIFICATION RULES

A few rules must be followed when creating a file specification:

1. The punctuation marks are required to separate the fields of the file specification.

2. Spaces are not allowed within a file specification.

3. The name chosen for each portion (except the version number) may contain digits or characters, but must begin with a character.

4. Each portion of the file specification is limited to a certain length:

   o Node: 1-6 characters

   o Device: 1-15 characters

   o Directory: 1-9 characters

   o File_name: 1-9 characters

   o File_type: Ø-3 characters

   o Version number: 1-5 digits

3.2.2 DIRECTORIES AND SUBDIRECTORIES

A directory file is a special kind of file. Directory files contain a list of names of other files. They are used by the system to access the other files. Directories reside on disk volumes. Normally, one directory file is created for each user on a system. The name of this file is often the same as the user's last name.

A master directory file, named ØØØØØØ.DIR, resides on each volume. This master file contains a list of the names of the top-level directory files on the volume (usually the files whose names correspond to user names).

For example, a volume could contain the directory files for BROWN, SMITH, BLACK, and JONES. When the ØØØØØØ.DIR directory file is listed, all of these names are seen:

```
$DIRECTORY NODEA::DRA1:[ØØØØØØ]

BLACK.DIR;1   BROWN.DIR;1   JONES.DIR;1   SMITH.DIR;1
```

Several conclusions can be drawn from this example:

1. Even though the name of the master directory is ØØØØØØ.DIR, to specify the name of the directory in the command, the syntax [ØØØØØØ] must be used. This is true of all directory files. Their names are in the form name.DIR, but they must be specified as [name] in a file specification.

2. The DIRECTORY command always outputs file names in alphabetical order.

3. Directory files are always version 1.

The ØØØØØØ.DIR file is a list of files which are directory files themselves. Each of these directory files should contain a list of files, some of which could be directories. The directories listed in the master file are called top-level directories. The directories listed in top-level directories are called subdirectories. Subdirectories are directory files which contain a list of file names, some of which can be directories. Directories listed in a subdirectory are also called subdirectories.

Directory files and subdirectories can be better understood through the use of a tree diagram (like a family tree), as seen in Figure 3-1.

```
                 NODEA
                    |
                    |
       ------------------------------
        |             |             |
      DRAØ          DRA1          DRA2
        |             |             |
    ØØØØØØ.DIR;1   ØØØØØØ.DIR;1   ØØØØØØ.DIR;1
        .             |              .
        .             |              .
        .             |              .
                      |
    ---------------------------------------
     |           |          |             |
BLACK.DIR;1  BROWN.DIR;3  JONES.DIR;5  SMITH.DIR;41
     .           .          .             |
     .           .          .             |
     .           .          .             |
                                          |
                     -------------------------------
                      |              |             |
               PROJECT1.DIR;1   FILE.DAT;1Ø    TEST.FOR;4
                      |
        --------------------------
        |             |          |
   SHIPSPD.BAS;2  DATA.DAT;6  PROJNOTES.DIR;7
                                 |
                        -----------------------
                        |                     |
                NOTESDATA.DAT;3         SHIPNOTES.DAT;9
```

Figure 3-1

In this figure, the files listed reside on the volume mounted in the disk drive, DRA1. The DRA1 device, as well as the DRAØ and DRA2 devices are connected to the system with the node name NODEA. Each volume contains a master directory.

The master directory on the volume mounted in the DRA1 device contains four top-level directories: BLACK.DIR, BROWN.DIR, JONES.DIR, and SMITH.DIR. The SMITH.DIR directory file (shown in figure) contains one directory file, PROJECT1.DIR. PROJECT1.DIR, a subdirectory of SMITH.DIR, contains a directory file, PROJNOTES.DIR.

Notice that directories also contain other kinds of files.

The number of directory files which may be listed in any directory file is not limited. Therefore, SMITH.DIR could contain the names of more than one subdirectory, and each subdirectory file could contain the names of several other subdirectory files. However, only seven levels of directories may be defined from the top. (SMITH.DIR is a top-level or first-level directory. PROJECT1.DIR is a second-level directory. PROJNOTES.DIR is a third-level directory.)

3.2.3 PURPOSE OF DIRECTORIES AND SUBDIRECTORIES

The major reason directories and subdirectories are created is to logically separate information on a volume. When users are separated from each other through the use of top-level directories, each user appears to own a portion of the volume for storage of information. VMS supports a protection scheme which can be used to prevent other users from accessing files. This protection can be used to protect an entire directory from access, or to protect only a few of the files in the directory.

In some situations, one user could be working on several projects, each requiring several files. Subdirectories can be used to separate the files belonging to one project from files belonging to another.

Subdirectories become very useful for a frequent user because directory listings can be very long. OpWhen information is separated, each directory is smaller and easier to work with. Any user can create a subdirectory with their own directory structure with the CREATE/DIRECTORY [name] DCL command.

3.2.4 SPECIFYING FILES IN SUBDIRECTORIES

The system assumes that a master directory is stored on each volume. When a file specification is input, the system searches the master directory for the directory name input. If the directory name is listed in the master file, the system searches the directory file for the file_name.

If a file is stored in a subdirectory, the file_name is not listed in the top-level directory file; rather, it is listed in the subdirectory file. Therefore, the system must be given the name of the subdirectory file to search. In a file specification, this is done in the [DIRECTORY] portion by specifying the top-level directory name followed by a period. After the period, the subdirectory name is specified. If the file_name is listed in a second-level subdirectory, the [DIRECTORY] portion will contain two names. For example, to specify DATA.DAT in the subdirectory PROJECT1.DIR (see Figure 3-1), the following file specification can be used:

```
NODEA::DRA1:[SMITH.PROJECT1]DATA.DAT
```

If the file_name is listed in a third-level subdirectory, the top-level name and the second-level name must be specified first to provide a search path for the system. For example, to specify NOTESDATA.DAT in the subdirectory PROJNOTES.DIR (see Figure 3-1), the following specification can be used:

```
NODEA::DRA1:[SMITH.PROJECT1.PROJNOTES]NOTESDATA.DAT
```

3.2.5 DEFAULTS

Most users never have to input the complete file specification to uniquely identify a file to the system. This is because the system supplies several fields of the specification if the user does not specify them. These supplied fields are called **defaults.** The system stores some default values as part of the user's process. It is possible to default any field of the specification except the file_name. However, fields may be defaulted only under certain conditions:

- The **node** (the name of the system) may be defaulted if the file resides on a device attached to the system where the user is currently working.

- The name of one device where the user's top-level directory file is stored is recorded in the UAF for that user. If a device is not included in the specification, the name of this device (the default) is supplied.

- The name of the user's top-level directory is normally recorded in the UAF. The system supplies this directory name if the user does not specify a directory.

- The name of each file is unique, so the user must always supply a file_name. The system does not supply a default.

- The kind of information stored in each file should be indicated by the **file_type.** Users may choose any file_type desired, but if the standard file_types are used, certain system programs will supply the file_type field of the specification. For example, the PRINT and TYPE programs will always supply the file_type of LIS. However, if the user desires to print a file of type FOR, the file_type of FOR should be included in the file specification.

  Some system programs which accept input files and produce output files will assume one file_type for files input to them, and supply a different file_type for the output files. For example, the FORTRAN compiler assumes input files have the file_type of FOR, and supplies the OBJ file_type for files output.

- The **version number**, as previously stated, is set to 1 by default when the file is created. As modified versions are created, each is given a new version number. Version numbers are incremented by 1 automatically. A user may assign any version number to a file or allow the system to assign numbers. System programs choose the version with the highest number by default if no number is given.

Defaulting can be seen in the following example:

Joe Brown is working
 on a system whose name is NODEA,
 where his files are stored on a device named DRAØ
 in the top-level directory, [BROWN].
He is working with a file, TESTPRGM,
 whose file type is LIS.
This is the third version of the file, and the other two
 versions are also residing in the [BROWN] directory.

The program invoked by the PRINT command assumes all files input are of the type LIS. To print the file, Joe Brown can use any of the following commands:

```
$ PRINT NODEA::DRAØ:[BROWN]TESTPRGM.LIS;3

$ PRINT DRAØ:[BROWN]TESTPRGM.LIS;3

$ PRINT [BROWN]TESTPRGM.LIS;3

$ PRINT TESTPRGM.LIS;3

$ PRINT TESTPRGM.LIS

$ PRINT TESTPRGM
```

3.2.6 CHANGING DEFAULTS

Users can change the defaults recorded in their process. The SET NODE command is used to change the default node name to access another system connected by DECnet to the current system. The SET DEFAULT command can be used to change either the device name and/or the directory name. The new device name must correspond to an actual device on the system, and the new directory name must correspond to an existing directory.

For example, the device and directory names recorded in the UAF entry for Joe Smith are DRAØ and [SMITH], respectively (see Figure 3-1). When Joe logs in, the system sets his default to DRAØ:[SMITH]. To compile DRAØ:[SMITH]TEST.FOR;4, Joe only has to enter the command:

```
$FORTRAN TEST
```

If Joe wants to print DATA.DAT in the subdirectory PROJECT1.DIR (see Figure 3-1), the following command can be entered:

```
$PRINT [SMITH.PROJECT1]DATA.DAT
```

If Joe wants to work with several files for a while in that subdirectory, he could change his default:

```
$SET DEFAULT [SMITH.PROJECT1]

$PRINT DATA.DAT
```

Notice that Joe only has to enter the file_name and file_type after the default has been changed, since the default directory name is now [SMITH.PROJECT1].

To change the default directory name back to [SMITH], the following command can be used:

```
$SET DEFAULT [SMITH]

$PRINT [SMITH.PROJECT1]DATA.DAT

$PRINT DATA.DAT
  error message
```

Notice that if Joe tries to print DATA.DAT now, the complete directory specification must be given, or an error message results.

3.2.7 WILDCARDS

To list the names of all files in a directory, the DIRECTORY command is used:

```
$DIRECTORY [SMITH]
```

To list the names of all files whose type is FOR in a directory, a wildcard, *, may be used instead of any particular file_name:

```
$DIRECTORY [SMITH]*.FOR
```

To list the names of all files whose names begin with G in a directory, the wildcard may also be used:

```
$DIRECTORY [SMITH]G*.*
```

To list all versions of a file:

```
$DIRECTORY [SMITH]FILE.DAT;*
```

This wildcard may be used in the directory, file_name, file_type, and version number portions of the file specification. The purpose of the wildcard is to save time and effort on the part of the user.

Another useful wildcard is the period (.). The period is used within the [DIRECTORY] portion of the file specification:

```
$DIRECTORY [SMITH.PROJECT1]

$DIRECTORY [.PROJECT1]
```

By using the period, the user did not have to enter the name SMITH. The system takes the current default value for the directory name, and includes it before the period. Then, the completed file specification is used to search for the requested file.

Therefore, if the default value is [SMITH.PROJECT1], the files in the subdirectory PROJNOTES, can be listed using:

```
$DIRECTORY [SMITH.PROJECT1.PROJNOTES]
```

or

```
$DIRECTORY [.PROJNOTES]
```

Two other wildcards may be used with the directory portion as well; the ellipsis (...), and the hyphen (-). The meaning of the ellipsis is to search down through the directory structure. So, to list all files in the current directory and all subdirectories:

```
$DIRECTORY [...]
```

The hyphen is used to mean back up one directory level. So, if the default is set to [SMITH.PROJECT1], and the user wanted to list the files in [SMITH]:

```
$DIRECTORY [-]
```

Wildcards may be used in conjunction with directory names. So, to list the files in the PROJECT1 subdirectory and all files below it (assuming the default directory is [SMITH]):

```
$DIRECTORY [.PROJECT1...]
```

If the default is set to [SMITH.PROJECT1], and the user wanted to list all files in [SMITH] and all files in the rest of the structure:

```
$DIRECTORY [-...]
```

Other combinations may be used. Users should practice wildcards with the DIRECTORY command, as this command does not change anything. However, the wildcards are valid for use within most DCL commands requiring file specifications as parameters.

3.3 DECIPHERING ERROR MESSAGES

When a problem occurs in a program, utility, or DCL command, an error message is displayed. The error message contains four parts and appears in the following format:

```
%FACILITY-L-IDENT,TEXT
```

%FACILITY is the name of the system program or utility that generated this error message (for example, DCL).

L is the level of the error. There are five levels of errors:

- S - Successful. No error is reported. Usually, no message is output if a program is successful.
- I - Informational. No error, but the program outputs some information needed by the user. Often, these types of messages do not appear in the above format. Informational messages usually consist of text only.
- W - Warning. The program may have completed successfully, or there may have been an error. The user should check to see if the desired task has been completed.
- E - Error. The program has encountered an error. The program outputs the message and attempts to continue if possible.
- F - Fatal or severe error. The program is not able to recover from this error and continue. The program is aborted.

IDENT is a code word that is an abbreviation of the message text.

TEXT is a descriptive message that tells the user what the problem is.

The following example shows the error message which results when a command unknown to the DCL interpreter is entered after the $ prompt.

```
    $SDDD
    %DCL-W-IVVERB,unrecognized command
     \SDDD\
    $
```

The error message is a warning, output by the DCL interpreter. The incorrect command is also echoed. (Most messages include the echoing of incorrect input in some format; not always enclosed in backslashes.)

Some errors are detected by more than one utility, so several messages may be output. Usually, the first message contains the most pertinent information, but the others can be helpful.

For example:

```
 $PRINT FILE.DAT
 %PRINT-W-OPENIN, error opening DRA0:[BROWN]FILE.DAT as input
 -RMS-E-FNF, file not found
```

In this example, the file to be printed could not be found by RMS, so the PRINT program could not open it to print it. To correct this error, the user should create the file or enter the name of an existing file.

The user should ask the following questions when an error is received, because the problem is usually a common one:

- o Is every part of the command spelled correctly?
- o Does the command exist (is it a valid DCL command)?
- o Were the options, qualifiers,and/or parameters chosen from the list displayed for the command by the HELP program?
- o Was the command entered correctly (i.e., are the options, qualifiers, and parameters, if any, in the correct order)?
- o Is the user allowed to use the command?
- o Is the user trying to access a non-existent or restricted piece of hardware or software?

CHAPTER 4

CREATING AND MANIPULATING FILES

4.1 CREATING FILES WITH EDT

EDT is the DIGITAL standard editor for text files. Files containing text can be created and modified using the EDT editor. The following command is used to invoke the editor:

```
    $EDIT file_specification
```

Usually, the file_name and file_type are sufficient for the file specification. If the user desires to create a file on a different device or in a different directory than the current default values specify, the device and directory portions of the file specification will have to be included.

When a file is created, the file is assigned the version number of 1. If the editor is being used to modify an old file, the editor will open the file of the name given which has the highest version number.

Some examples:

```
    $EDIT FILE.DAT                 (uses defaults)

    $EDIT DRAØ:[SMITH]FILE.DAT;1   (no defaults used
                                    except system name)
```

To create a file in a subdirectory, the same kind of command is used:

Method one:

```
    $SET DEFAULT DRAØ:[SMITH.PROJECT1]
    $EDIT DATA.DAT
```

Method two:

```
    $EDIT DRAØ:[SMITH.PROJECT1]DATA.DAT
```

When the carriage return is pressed after the command is input, the editor is invoked. The EDT editor outputs a message and a prompt. The EDT prompt is an asterisk, *.

The EDT editor is capable of being in one of two modes, line mode and character mode. The * signals the user that EDT is in **line mode**, and is ready to accept line mode commands. (Note: DCL commands can not be input after the * prompt.)

One line mode command is CHANGE, (can be abbreviated to C). When this command is input, the mode is changed to character **mode**. No prompt is output for character mode, and the editor will only accept character mode commands. (Note: Neither DCL commands nor EDT line mode commands are accepted when there is no prompt.)

A Computer-Based course is available that will teach you how to use the features of EDT. Contact your system manager to see if this course is available on your system.

4.1.1 EDT LINE MODE COMMANDS

Since character mode is so easy to use on video terminals, most line mode commands are only used on hardcopy terminals. People working on video terminals will normally use the CHANGE (to enter character mode), EXIT, QUIT, and SUBSTITUTE line commands.

In line mode, the EDT editor numbers each line so it can be identified. Line numbers begin at Ø and the normal increment is 1. However, fractional numbers are used also. For example, if a line is inserted between lines 1 and 2, the new line is given the number of 1.5. When too many lines have been inserted, numbers are not assigned to the new lines. At this point, the user can enter the RESEQUENCE command to renumber the file in increments of 1 (or some other chosen increment).

To indicate a line in a line mode command, the number of the line should be specified. To indicate several lines, a range can be specified by entering the number of the first line, followed by a colon and the number of the last line. For example, to DELETE lines 2 through 1Ø (inclusive), the range is specified as 2:1Ø. To indicate the entire file, as often happens with the SUBSTITUTE command, the symbol %WH (or %WHOLE) can be entered (see Table 4-1 for an example).

All EDT line mode commands are terminated by the input of a carriage return. All commands can be abbreviated (see Table 4-1) except the QUIT command.

Table 4-1 lists a subset of line mode commands. The EDT editor has on-line HELP, so help can be obtained on each of the commands listed.

Table 4-1 Subset of EDT line mode commands

Command	Function	Example(s)
CHANGE	To change to character mode	*CHANGE or *C
COPY	To copy a line or a group of lines from one area of the file to BEFORE another line in the file	*COPY 1Ø TO 1ØØ *CO 1:5 TO 8
DELETE	Delete a line or group of lines	*DELETE 1Ø *D11:25
EXIT	Exit from the editor, saving all changes	*EXIT or *EX
HELP	Obtain help on all line mode commands	*HELP or *H
INSERT	Add text to the file. Editor inserts BEFORE current position or BEFORE line number specified. No prompt is output while inserting. To return to the * prompt, press <CTRL-Z>.	*INSERT new text <CTRL-Z> *I5 other new text <CTRL-Z> *
MOVE	Move a line or lines from one area of the file to BEFORE a line in another area	*MOVE 1Ø TO 5 *MO 3:4 TO 11
QUIT	Exit from the editor without saving any changes	*QUIT
REPLACE	Delete a line or group of lines and enter Insert mode to add text	*REPLACE 1Ø or *R1Ø 1 line deleted new text added <CTRL-Z> *
RESEQUENCE	Renumber all lines in the file in increments of 1	*RESEQUENCE *RES
SUBSTITUTE	Substitute a new piece of text for an old piece	*SUBSTITUTE/old/new/%WH *S/text/newtext/1Ø:2Ø

4.1.2 EDT KEYPAD MODE COMMANDS

Character mode in the EDT editor is easy to learn, fast to use, and powerful. No prompt is output, because all commands are based on the current position of the cursor (the flashing light on the screen).

In character mode, the user is always inserting. Whenever a character is entered from the main keyboard, it is echoed on the terminal and becomes part of the file. New lines are created by pressing the carriage return. Commands are entered by using the keypad to the right of the keyboard. Character mode commands are terminated when they are input. (A carriage return does not mean 'end of command' in character mode.)

Each key on the keypad means something different to the editor. Figure 4-1 shows the layout of the keypads for the VT52 and VT1ØØ. The commands available on each terminal are similar, but the keypad layout is different. Most users cut out a copy of one of these diagrams to paste to the front of the appropriate terminal for reference.

The easiest way to learn how to use character mode is by using it. The following list of character mode commands should be practiced on a practice file until the user is familiar with them.

MAJOR KEYS

- GOLD - used in conjunction with other keys. Normally, the command associated with a key is the command listed at the top of the square corresponding to the key in Figure 4-1. To invoke the commands at the bottom of the square, press GOLD, and then press the key. For example, the DEL C key deletes a character. Pressing GOLD and the DEL C key will undelete a character.

- HELP - will output a picture of Figure 4-1 for the current terminal and allow the user to obtain HELP for any of the keys on the keypad.

- ADVANCE - When pressed, causes the cursor to be in advance mode (the default). All commands used to move the cursor will move it in a forward direction, towards the end of the file.

- BACKUP - When pressed, causes the cursor to be in backup mode. All commands used to move the cursor will move it in a backward direction, towards the beginning of the file.

Commands affected by ADVANCE or BACKUP

- SECT - moves the cursor several lines at a time
- LINE - moves the cursor one line at a time
- WORD - moves the cursor one word at a time
- CHAR - moves the cursor one character at a time
- EOL - moves the cursor to the end of a line

Commands not affected by ADVANCE or BACKUP

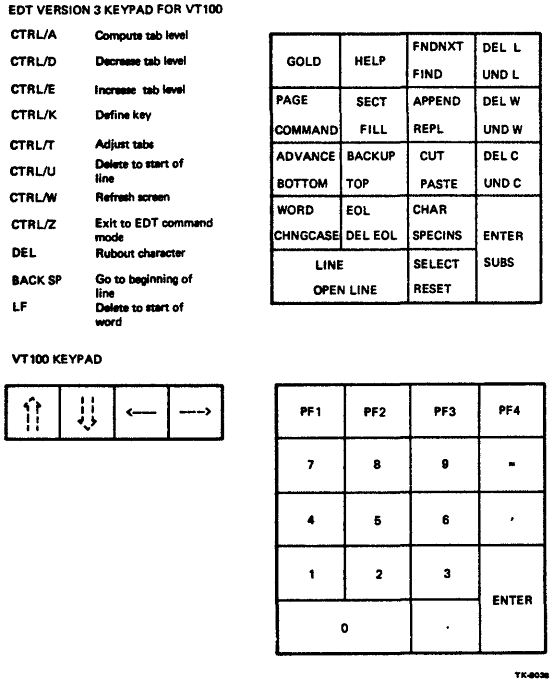
- DEL CHAR - deletes the character at the cursor position

  (DELETE - not on the keypad, but on the regular keyboard, deletes one character to the left of the cursor as usual)

- DEL WORD - deletes the word to the right of the cursor
- DEL LINE - deletes the line to the right of the cursor (including the carriage return and line feed)

Note that when the DEL CHAR, DEL WORD, and DEL LINE keys are used, the deleted text is saved in a temporary buffer so the user can UNDelete the text. This is useful in the case of an accident, where text is unintentionally deleted. It is also useful when the user wants the same line of text to be placed in several places in the file. The user can delete the line, undelete it, and then move to the other places, undeleting the line wherever it is needed. However, these buffers only hold one value (i.e., one line, one word, or one character) at a time. They are overwritten by newly deleted values.

If the user would like to save several lines of text in a buffer, to be placed in another place or several places in the file, the CUT and PASTE keys should be used. To save the text, the user should position the cursor at the beginning of the text and press SELECT. Then, the user should position the cursor after the end of the text and press CUT. The selected text will be removed from the file and placed in a buffer. Therefore, the text is deleted. The user could stop here, or replace the text elsewhere in the file by moving the cursor to the desired position and pressing PASTE. The text will be inserted before the current position of the cursor when PASTE is pressed. (Note that the GOLD key must be pressed before the PASTE key to enter the PASTE command.)



EDT VERSION 3 KEYPAD FOR VT100

CTRL/A	Compute tab level
CTRL/D	Decrease tab level
CTRL/E	Increase tab level
CTRL/K	Define key
CTRL/T	Adjust tabs
CTRL/U	Delete to start of line
CTRL/W	Refresh screen
CTRL/Z	Exit to EDT command mode
DEL	Rubout character
BACK SP	Go to beginning of line
LF	Delete to start of word

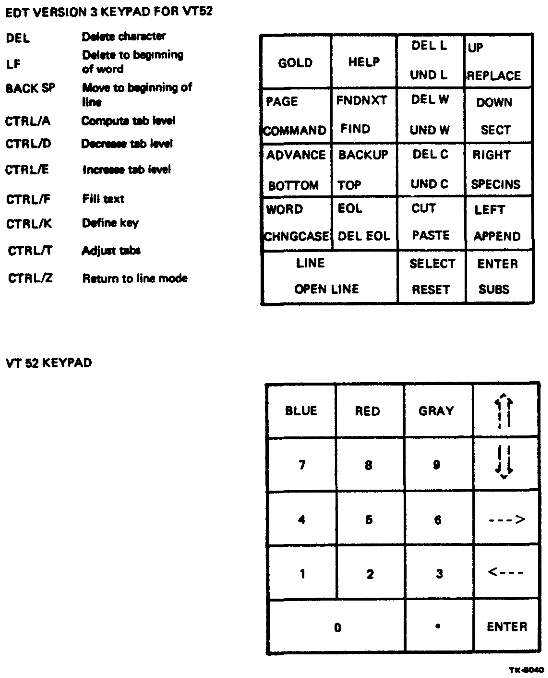
GOLD	HELP	FNDNXT FIND	DEL L UND L
PAGE COMMAND	SECT FILL	APPEND REPL	DEL W UND W
ADVANCE BOTTOM	BACKUP TOP	CUT PASTE	DEL C UND C
WORD CHNGCASE	EOL DEL EOL	CHAR SPECINS	ENTER SUBS
LINE OPEN LINE		SELECT RESET	

VT100 KEYPAD

↑	↓	<——	——>

PF1	PF2	PF3	PF4
7	8	9	-
4	5	6	,
1	2	3	ENTER
0		.	

TK-8039



EDT VERSION 3 KEYPAD FOR VT52

DEL	Delete character
LF	Delete to beginning of word
BACK SP	Move to beginning of line
CTRL/A	Compute tab level
CTRL/D	Decrease tab level
CTRL/E	Increase tab level
CTRL/F	Fill text
CTRL/K	Define key
CTRL/T	Adjust tabs
CTRL/Z	Return to line mode

GOLD	HELP	DEL L UND L	UP REPLACE
PAGE COMMAND	FNDNXT FIND	DEL W UND W	DOWN SECT
ADVANCE BOTTOM	BACKUP TOP	DEL C UND C	RIGHT SPECINS
WORD CHNGCASE	EOL DEL EOL	CUT PASTE	LEFT APPEND
LINE OPEN LINE		SELECT RESET	ENTER SUBS

VT 52 KEYPAD

BLUE	RED	GRAY	↑
7	8	9	↓
4	5	6	--->
1	2	3	<---
0		.	ENTER

TK-8040

Other EDT character mode commands are used less frequently. Information about them can be obtained through the HELP facility.

4.1.3 RECOVERING FROM A SYSTEM FAILURE

Recovering from a system failure during an edit session is not difficult with the EDT editor. While the user is editing, EDT is creating a journal file. This journal file contains a list of all commands entered since the beginning of the session. After the system is running again, users can recover all edits done by using the command:

```
$EDIT/RECOVER file_specification
```

The user should specify the name of the file which was being edited at the time of the system crash. The EDT editor will read the latest version of that file as input, and use the commands listed in the journal file of the same name (name.JOU) to reconstruct the work done. During recovery, the editor will actually repeat the work done previously by the user. Users should not touch the keyboard until the editor is done and a prompt (if they were in line mode) appears. If the system crashed while the user was in character mode, the user should wait until the cursor stops moving around. After the editor completes the journal file's list of commands, it will accept commands from the user. (Note: A journal file will also be created if the user exits the editor incorrectly (i.e. with a CTRL-Y).)

4.2 FILE MANIPULATION WITH DCL COMMANDS

Several DCL commands are useful for moving, copying, printing, and obtaining information about files. Table 4-2 contains the most commonly used DCL commands for these purposes.

The * wildcard can be used with any of these commands in place of one or more fields of the file specification. Notice that most commands will prompt for missing information. This is especially useful for the COPY command, as shown.

More information about any of the commands in Table 4-2 can be obtained through the use of the HELP command.

Table 4-2 Commonly used DCL commands for file manipulation

Command	Function	Example
DIRECTORY	Used to obtain information about files. The /FULL qualifier is used to obtain more information.	$DIRECTORY $DIRECTORY/FULL
COPY	Used to copy information stored in one file to another file. The second file usually has a different file specification. (Result is two files containing the same information)	$COPY $_from:FILE.TXT $_to:DATA.DAT
RENAME	Used to change the name of a file.	$RENAME $_from:DATA.DAT $_to:TEST.FOR
PRINT	For printing a file on the system default printer designated by the system manager.	$PRINT BUDGET.FOR
TYPE	For outputting the contents of a file to the terminal.	$TYPE FILE.BAS
DELETE	To delete a file. Requires a version number.	$DELETE NAME.DAT;3
PURGE	To delete all but the latest version of any or all files in a directory.	$PURGE $PURGE FILE.DAT

CHAPTER 5

PROGRAM DEVELOPMENT

5.Ø INTRODUCTION TO PROGRAM DEVELOPMENT

VAX/VMS provides a number of tools that significantly decrease the time spent developing VAX-11 programs. These tools include:

- Interactive Text Editor (EDT)
- Programming Languages
- Linker
- Librarian
- Common Run-Time Library
- Symbolic Debugger
- Record Management Services

The editors, programming languages, and linker, are utilities that are used to prepare a source program for execution. The symbolic debugger is used to detect errors in executable programs (programs that do not appear to contain errors when compiled/assembled and linked, but, nevertheless, fail to produce correct results).

The librarian enables storage of frequently-used segments of code, such as procedures or functions, in specially indexed files called libraries. Procedures or functions stored in a library can be referenced in a program. The linker combines the code from the library with the user's source code to produce an executable image.

For the MACRO language, definitions (macros) can be stored in a different type of library. Libraries containing macros can be accessed by the assembler to include a specific macro in the program.

The Run-Time Library is a system library containing a large number of predefined routines that can be called from user programs (such as routines to manipulate strings or generate random numbers). The MACRO programmer will find some of the I/O routines to be especially useful, while high-level language programmers will probably use the math or bit manipulation routines more often. Help can be obtained on-line for most of the Run-Time Library routines by entering the DCL command, HELP RTL, and specifying one of the categories listed as the subtopic.

This chapter begins with a discussion of program development in general, followed by sections on each of several VAX-11 programming languages (MACRO, FORTRAN, PASCAL, BASIC, and COBOL). Those sections contain a discussion of the VAX-11 specific conventions regarding that language, a sample program, and a debug session using the sample program.

5.1 PROGRAM DEVELOPMENT ON VAX/VMS

To develop a program written in a programming language, the following sequence of steps must be completed:

1. Create a text file with an editor which contains statements written in a programming language.

2. Compile/Assemble the source program.

3. Link the compiled program to create an executable image.

4. Test the program.

5. Debug (make corrections to) the source program and repeat steps 2 through 5 until the program executes properly.

These steps are explained in detail on the following pages.

1. Create a text file which contains the source statements of your program.

   The file_type should be related to the language being used. Two reasons that this is important is that it helps you tell a source program in one language from another, and the compilers will search for certain default file_types as shown below.

Language	Default File_Type
MACRO	.MAR
FORTRAN	.FOR
BASIC	.BAS
PASCAL	.PAS
PLI	.PLI
COBOL	.COB

   The entire program may be entered in one text file, or several files may be created. Usually, if several files are created, the code representing the main program is entered in one file, and the subprograms referenced are each placed in separate files. There is no rule stating a limit on the number of subprograms per text file. However, if each is in a separate file, they are more accessible to other programs.

2. Compile/assemble the text file to produce a file containing object code.

   The compiler/assembler translates the source statements of each input file into executable code, producing one or more object files of type .OBJ.

   To compile/assemble the code, the command related to the language must be used:

Language	Compiler/Assembler Command
MACRO	$MACRO file_specification
FORTRAN	$FORTRAN file_specification
BASIC	$BASIC file_specification
PASCAL	$PASCAL file_specification
PLI	$PLI file_specification
COBOL	$COBOL file_specification

   File_types other than the defaults listed earlier must be included in the file_specification. Otherwise, the appropriate file_type will be provided by the command used.

More than one input file may be listed as parameters. If input file_specifications are separated by (,) commas, a separate object file is created for each input file. If they are separated by (+) plus signs, one object file is created containing the code from all input files.

If syntax errors are found in the source code, an appropriate message will be output at the user's terminal. The DCL HELP command can be used to understand errors output by the compilers for FORTRAN, BASIC, and COBOL by entering HELP language ERROR.

When the error is understood, an editor should be used to correct the source code, and the new version of the text file should be submitted to the compiler/assembler for translation.

Many qualifiers can be used in conjunction with the compiler or assembler command. The DCL HELP command can be used to obtain information about qualifiers by entering the 'HELP language_name' command. Most compilers will take the following qualifiers with the compile command. You should check the user guide for the specific language for information on other qualifiers.

Qualifier	Use
/LIST	The most commonly used qualifier that causes a listing file to be produced as well as the object file. The file is useful when trying to debug the program.
/CROSS_REFERENCE	The cross reference qualifier tells the compiler to generate a cross reference listing. This type of list contains program symbols, their class, and the program lines in which they are referenced.
/DEBUG	The debug qualifier tells the compiler to provide information to the symbolic debugger and the system run-time error traceback mechanism.

For example, to compile a BASIC program, called SAMPLE, and obtain a list of the program as well as cross referenced listing of program variable you would type:

```
BASIC/LIST/CROSS_REFERENCE SAMPLE
```

3. Link the object file or files to produce an executable image.

   The linker assigns virtual addresses to the lines of executable code in each input file, and resolves references to symbols between modules. The linker also searches personal and system libraries for external procedures and functions that cannot be found in the input files specified.

   To link the object file(s), the VAX-11 Linker is invoked using the DCL command, LINK. The names of the files to be linked, such as object code files or modules from libraries, can be specified following the command. Names should be separated by commas. The linker assumes the file_type of input files is .OBJ.

   The file output by the linker contains executable code, and is assigned the file_type of .EXE.

   If the linker is unable to resolve certain symbols or to locate certain subprograms, it displays an appropriate error message. Linker errors usually indicate one of two problems:

   o A subprogram was referenced but not included in the list of input files

   o A subprogram/variable was not defined/referenced properly in the program

   Linker errors and recommended solutions are described in the VAX-11 Linker Reference Manual.

   Several qualifiers are available for use with the linker command (enter HELP LINK). Cross-reference listings, maps, and other information can be written to files or to the terminal by using these qualifiers. The information produced is most useful to the more advanced programmer, and will not be discussed in this document. The following table shows some of the most common LINK command qualifiers.

Qualifier	Use
/MAP	This qualifier produces a file containing a list of the symbols and data used in the program and their locations in memory.
/CROSS_REFERENCE	This qualifier produces a cross reference list of each global symbol used in the program, its value, the name of the first module in which is defined, and the name of each module in which it is referenced.
/DEBUG	The qualifier causes the linker to: (1) Generate a Debug Symbol Table (2) Gives control to the debugger when the image is run.

The following example illustrates the use of the LINK command to create an executable image of the program SAMPLE and creating a map file.

```
LINK/MAP SAMPLE
```

4. Test the image produced from the linker.

   To execute a program, enter the DCL command, RUN, followed by the name of a single executable image file. The run command assumes the file_type of the input file is .EXE.

   Users should not attempt to execute a program if compiler and linker errors have not been corrected.

   Errors output at run-time could indicate syntax problems not identified by the compiler/assembler or linker. Other run-time errors could be output by procedures referenced by the program, such as system routines. Some errors output by system routines are documented on-line. To look at a description of these errors, enter the DCL command, HELP ERROR, and enter the appropriate facility code as the subtopic. Information on other errors can be obtained by entering HELP ERROR SYSTEM error_code.

   If all obvious errors have been corrected, errors output at run-time can indicate logical errors. A logical error occurs because the organization of the statements in the program does not do the intended job. A logical error could produce error messages, or, simply, the wrong result. Results should be checked carefully. If the program receives input from the user, it should be executed several times with various types of input to be sure it does the required job in all given situations.

   To correct the program, the user must debug it to find out where the error is occurring. When the error is found, the source program must be modified and submitted to the compiler/assembler and linker again. Then the new executable file can be executed to see if the error was corrected.

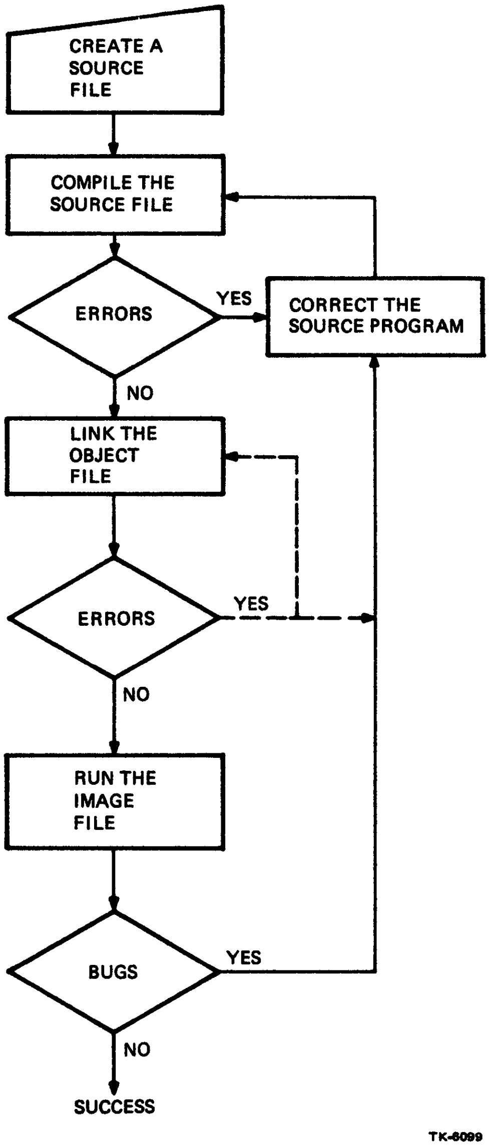
5. Debug the program to correct errors.

   To find the cause of a logical error, the user must examine the program carefully, looking at the source code one line at a time. Lists of variables and their contents should be kept on paper, as well as comments on loops and output to peripherals. Often, in larger programs, the problem can be isolated to a particular area of the program, saving the user the time of looking at every line.

   If the problem can be isolated, or the program is not very large, examining a program using paper is not difficult, and errors can be easily found. As larger programs are written, involving more I/O and more variables and more loops, debugging becomes more complicated.

   The VAX-11 Symbolic Debugger is provided to simplify the user's debugging job. Symbolic debugger commands implement the same debugging techniques used on paper.

   The flowchart in Table 5-1 summarizes the program development steps. Although the flowchart in the table uses a FORTRAN program, the flowchart can be used for a program written in any programming language.



Developing a Program

5.2 LOGICAL NAMES

If a file_specification or device name is included in the source file for a program, the program is said to be file dependent or device dependent. When the program is dependent, the file or device must exist when the program is executed, and the program always outputs to or inputs from the file or device specified.

File and device independence can be achieved through the use of logical names. A logical name is created by the DCL command ASSIGN, and can be used in a program instead of the file or device name. The ASSIGN command assigns a specified logical name to a specified device or file name (called the equivalence name). When the logical name is encountered in a program, the system translates it into the equivalence name. The general forms of the DCL ASSIGN command are:

```
ASSIGN device: logical name
ASSIGN file_specification logical name
```

The example below illustrates the use of the ASSIGN command to make a program device and file independent.

```
      PROGRAM1                           PROGRAM2

File dependent program          File independent program

 writes to particular file,      writes to logical name,
     FILE.DAT                            OUTPUT_FILE


Execution of PROGRAM1:          Execution of PROGRAM2:

 $RUN PROGRAM1                   $ASSIGN GENERAL.DAT OUTPUT_FILE
 $TYPE FILE.DAT                  $RUN PROGRAM2
  contains output                $TYPE GENERAL.DAT
  from 1st execution              contains output from
 $RUN PROGRAM1                    1st execution
 $TYPE FILE.DAT                  $ASSIGN OUTPUT.DAT OUTPUT_FILE
  contains output                $RUN PROGRAM2
  from 2nd execution             $TYPE OUTPUT.DAT
                                  contains output from
                                  2nd execution
```

Notice that PROGRAM1 always outputs to FILE.DAT, whereas PROGRAM2 can send output to a different file each time it is executed. (The assignment command must be executed prior to the execution of the program.)

Several logical names are provided by the system, and are stored in the user's process logical name table. To look at the table, use the DCL command SHOW LOGICAL/PROCESS. Table 5-2 lists some of the system-defined logical names commonly used in programs.

Table 5-2 System-defined logical names

Logical name	Equivalence name
SYS$INPUT	Default input device. For the interactive user, SYS$INPUT is equated to the terminal.
SYS$OUTPUT	Default output device. For the interactive user, SYS$OUTPUT is equated to the terminal.
SYS$DISK	Default user disk established at login time. Can be changed by SET DEFAULT command.
SYS$LOGIN	Default user disk and directory established at login time. Usually the top-level directory. Specified in the user's UAF entry by the system manager.

5.3 A SAMPLE PROGRAM -- GRADES

The GRADES program has been created in each language discussed in this chapter. The listing file for each language's implementation of GRADES is included in the section of the chapter discussing that language (following this section).

The GRADES program creates a file containing the names of students and the average of their grades for a particular course. The program obtains the names and grades from the user, computes the average of the grades, and outputs the results to the terminal and to a designated file. The logical name 'Course', created before the program is executed, is assigned to the name of the output file. For example:

```
$ASSIGN HISTORY.DAT Course
$RUN GRADES
```

In this example, the program GRADES is executed to compute the average of the grades for the students in the history class. The output file, HISTORY.DAT, is assigned to the logical name 'Course' before the program is executed. The program writes results to the logical name 'Course'.

5.3.1 NORMAL EXECUTION OF GRADES

A sample run of the GRADES program follows. The FORTRAN version was used in this example:

```
$ASSIGN ENGLISH.DAT COURSE
$FORTRAN GRADES
$LINK GRADES
$RUN GRADES

Student name? JOHN SMITH
Input grade (or 0 to end input): 45
Input grade (or 0 to end input): 80
Input grade (or 0 to end input): 99
Input grade (or 0 to end input): 0

Student: JOHN SMITH                  Average:    74.7

Are you done ? (Yes/No) N

Student name? MARY HAGERTY
Input grade (or 0 to end input): 82
Input grade (or 0 to end input): 69
Input grade (or 0 to end input): 94
Input grade (or 0 to end input): 0

Student: MARY HAGERTY                Average:    81.7

Are you done ? (Yes/No) N

Student name? HOSIAH HOWER
Input grade (or 0 to end input): 90
Input grade (or 0 to end input): 78
Input grade (or 0 to end input): 81
Input grade (or 0 to end input): 0

Student: HOSIAH HOWER                Average:    83.0

Are you done ? (Yes/No) Y
$
$
$TYPE ENGLISH.DAT

Student: JOHN SMITH                  Average:    74.7

Student: MARY HAGERTY                Average:    81.7

Student: HOSIAH HOWER                Average:    83.0
$
```

5.4 USING THE SYMBOLIC DEBUGGER

Three methods are available for invoking the Symbolic Debugger:

1. Including the debugger in the executable image.

   The debugger is included in the executable image if the /DEBUG qualifier is entered with the LINK command. When your program is subsequently executed, the debugger is automatically invoked, and the debug prompt is output. For example:

   ```
   $LINK/DEBUG filename
   ```

   Unless the /DEBUG qualifier is also included in the compiler command (/ENABLE=DEBUG with the MACRO command), local symbol tables will not be included. The symbol tables contain the names and addresses of various symbols and variables used in the program. If the user intends to examine the contents of variables, the tables should be included. Other debug commands, such as GO, STEP, or setting tracepoints, work without this information.

2. Halting the program and invoking the debugger with the DCL command $DEBUG.

   A program can be halted by entering <CTRL/Y> or <CTRL/C>. The debugger can then be invoked by entering the DCL command, DEBUG. In this case, the debugger does not have access to local symbols.

   This method can be used to halt a 'hung' program, one that will not run to completion. The debugger can be used to determine where the program is hung.

   This method can also be used for a program that is executing in the debugger already in case the user wants to input a debug command at a time when the debug prompt is not seen.

3. Running the program with the debugger.

   A program can be run with the debugger if the /DEBUG qualifier is included in the RUN command. Again, if the debug qualifier was not included with the compiler/assembler command, the symbol tables will not be included and the contents of variables can not be accessed.

Table 5-3 Major Symbolic Debugger Commands

Feature	Description	Command Format
Display values	Display variable contents using symbolic names	EXAMINE variable
Change values	Modify variable contents	DEPOSIT variable = value
Define symbols	Define symbolic names for later use	DEFINE symbol = value
Calculate values	Compute expressions using symbolic names	EVALUATE expression
Get help	Get help for any command	HELP [command_name]
Breakpoints	Suspend program execution at a specified point	SET BREAK at line #
Tracepoints	Monitor order of execution of program lines	SET TRACE at line #
Watchpoints	Suspend program execution when the content of a variable changes	SET WATCH variable
Test subroutines	Call and pass arguments to a subroutine	CALL sub_name [(arg,...)]
Execute program	- from a given point	GO [address]
	- for a specified number (n) of instructions or lines	STEP [n]
Debug routines	Make symbols from specified module available to debugger	SET MODULE module
	Define default module name for setting tracepoints and watchpoints on symbols whose names appear in more than one module	SET SCOPE module
Stop debugger	Leave debugger and return to DCL prompt	EXIT

Note: Fields enclosed in [] (brackets) are optional.

5.4.1 EXECUTION OF GRADES WITH THE DEBUGGER

Three examples of the GRADES program using the debugger follow. The FORTRAN version of the program was used. The syntax of most of the debug commands shown is the same for other languages. Therefore, these debug examples and associated comments should be read by all users. A listing of the FORTRAN program is provided before the examples.

Each of the languages mentioned earlier is discussed briefly in the sections following these examples. A listing of the GRADES program is included, followed by a discussion on using the symbolic debugger with that language.

A brief description of most of the commands used can be found in Table 5-3. The HELP facility in the debugger can be used to obtain more information. More discussion of some of the commands and their output is included with the examples.

Listing of Main Program

```
        PROGRAM GRADES

        CHARACTER STUDENT_NAME*30, DONE*4
        REAL AVERAGE

        OPEN (UNIT=1, FILE='Course', STATUS='New')

10       TYPE 20
20       FORMAT (/' Student name? ',$)
         ACCEPT 30, STUDENT_NAME
30       FORMAT (1A30)

           CALL Get_grades_compute_average (AVERAGE)

         TYPE 40,STUDENT_NAME, AVERAGE
         WRITE (1,40) STUDENT_NAME,AVERAGE
40       FORMAT (/' Student: ',A30,'Average: ',F10.1)

         TYPE 50
50       FORMAT (/' Are you done ? (Yes/No) ',$)
         ACCEPT 60, DONE
60       FORMAT (1A4)
           IF (DONE.NE.'Y' .AND. DONE.NE.'y') GOTO 10

        CLOSE (UNIT=1)
        END
```

Listing of Subroutine

```

        SUBROUTINE Get_grades_compute_average (AVERAGE)

        INTEGER ICOUNT
        REAL TOTAL, GRADE
         ICOUNT = 0
         TOTAL = 0

10       TYPE 20
20       FORMAT (' Input grade (or 0 to end input): ',$)
         ACCEPT 30, GRADE
30       FORMAT (F10.0)

          IF (GRADE.NE.0) THEN
            ICOUNT = ICOUNT + 1
            TOTAL = TOTAL + GRADE
            GO TO 10
          ENDIF

40        IF (ICOUNT.NE.0) AVERAGE = TOTAL/ICOUNT

        RETURN
        END
```

EXAMPLE 1 -- Setting watchpoints and breakpoints

The following three DCL commands compile, link, and execute the program GRADES. Because the /DEBUG qualifier was used in the language compile and LINK command the symbolic debugger will gain control of the program execution.

```
$FORTRAN/LIST/DEBUG GRADES
$LINK/DEBUG GRADES
$RUN GRADES

                  VAX-11 DEBUG Version 3.0-5

%DEBUG-I-INITIAL, language is FORTRAN, module set to 'GRADES'
```

The EXAMINE command of the debugger allows you to check the contents of variables in the program. The DEPOSIT command gives you the opportunity to alter the contents of variables. With the following debug commands the value of the variable DONE is examined and altered. The SET WATCH command sets a watchpoint on the variable which causes the debugger to display the old and new values of the variable whenever the contents of the variable is altered. The SHOW WATCH command causes the debugger to display the locations at which watchpoints have been established.

```
DBG>EXAMINE DONE
GRADES\DONE(1:4):
DBG>DEPOSIT DONE="YES"
DBG>EXAMINE DONE
GRADES\DONE(1:4):  YES
DBG>SET WATCH DONE
DBG>SHOW WATCH
watchpoint at GRADES\DONE(1:4) for 4. bytes.
```

The GO command causes the program to execute or resume execution at the point it was suspended.

```
DBG>GO
routine start at GRADES

Student name? JOE SMITH
Input grade (or 0 to end input): 6
Input grade (or 0 to end input): 7
Input grade (or 0 to end input): 0

Student: JOE SMITH                      Average:    6.5

Are you done ? (Yes/No) N
```

Because a watchpoint was established for the variable DONE the debugger displays the old and new contents of the variable.

```
write to GRADES\DONE(1:4) at PC 70649
         old value = YES
         new value = N
DBG>EXAMINE DONE
GRADES\DONE(1:4):  N
```

In the following section a breakpoint is set with the SET BREAK command at line 23. When the program resumes execution with the GO command the debugger will indicate the module name and the line number where the program is interrupted.

```
DBG>SET BREAK %LINE 23
DBG>GO
start at 70659
break at GRADES\%LINE 23
DBG>GO
start at GRADES\%LINE 23
Student name? GERALD HORNER
Input grade (or 0 to end input): 50
Input grade (or 0 to end input): 100
Input grade (or 0 to end input): 0

Student: GERALD HORNER               Average     75.0

Are you done ? (Yes/No) N
write to GRADES\DONE(1:4) at PC 70649
         old value = N
         new value = N
```

The CANCEL WATCH command is used to cancel watchpoints that have been set.

```
DBG>CANCEL WATCH DONE
DBG>GO
start at 70659
break at GRADES\%LINE 23
DBG>GO
start at GRADES\%LINE 23
Student name? MARY HAGERTY
Input grade (or 0 to end input): 9
Input grade (or 0 to end input): 9
Input grade (or 0 to end input): 0

Student: MARY HAGERTY                Average      9.0

Are you done ? (Yes/No) N
```

The CANCEL BREAK command cancels a single breakpoint or by using the /ALL qualifier cancels all breakpoints set in the program. Now that all watchpoints and breakpoints have been cancelled program execution will continue until normal program execution.

```
DBG>CANCEL BREAK/ALL
DBG>GO
start at GRADES\%LINE 23
Student name? HORACE O'TOOLE
Input grade (or 0 to end input): 8
Input grade (or 0 to end input): 0

Student: HORACE O'TOOLE                   Average      8.0

Are you done ? (Yes/No) N

Student name? CRAIG SMYTHE
Input grade (or 0 to end input): 10
Input grade (or 0 to end input): 10
Input grade (or 0 to end input): 0

Student: CRAIG SMYTHE                     Average     10.0

Are you done ? (Yes/No) Y
```

The following message and command indicates normal program termination. Control is then returned to the debugger. The EXIT command terminates the debugger and returns control to DCL.

```
Is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>EXIT
$
```

EXAMPLE 2 -- Setting tracepoints and single stepping

Before you can use the symbolic debugger you must first compile the program with the /DEBUG qualifier, then link the program again using the /DEBUG qualifier. When you run the program the symbolic debugger will automatically take control of the execution of the program.

```
$FORTRAN/LIST/DEBUG GRADES
$LINK/DEBUG GRADES
$RUN GRADES

                    VAX-11 DEBUG VERSION 3.0-5

%DEBUG-I-INITIAL, Language is FORTRAN, module set to 'GRADES'
```

The EXAMINE and DEPOSIT commands allow you to check the values of variables in a program and alter the contents of those variables. In this example the value of DONE was examined and its contents displayed. With the DEPOSIT command the variable DONE was altered to contain "YES".

```
DBG>EXAMINE DONE
GRADES\DONE(1:4):
DBG>DEPOSIT DONE ="YES"
```

A breakpoint is set at line 23 so that execution of the program will be interrupted. When the program is interrupted the debugger displays the DBG> prompt. Using the SET TRACE command tracepoints are set in the program. Tracepoints allow you to examine the order in which the statements are being executed.

```
DBG>SET BREAK %LINE 23
DBG>SET TRACE %LINE 8
DBG>SET TRACE %LINE 13
DBG>SET TRACE %LINE 19
DBG>GO
routine start at GRADES
trace at GRADES\%LABEL 10

Student name? JOE SMITH
trace at GRADES\%LINE 13
Input grade (or 0 to end input): 6
Input grade (or 0 to end input): 7
Input grade (or 0 to end input): 0

Student: JOE SMITH                          Average:     6.5
trace at GRADES\%LINE 19

Are you done ? (Yes/No) N
break at GRADES\%LINE 23
```

While the GO command causes the program to execute until a breakpoint is reached or the program terminates normally, the STEP command causes the debugger to execute one single statement.

```
DBG>STEP
start at GRADES\%LINE 23
stepped to GRADES\%LABEL 10
DBG>STEP
start at GRADES\%LABEL 10
trace at GRADES\%LABEL 10

Student name? MARY HAGERTY
trace at GRADES\%LINE 13

Input grade (or 0 to end input): 100
Input grade (or 0 to end input): 50
Input grade (or 0 to end input): 0

Student: MARY HAGERTY                    Average:   75.0
trace at GRADES\%LINE 19

Are you done ? (Yes/No) N
break at GRADES\%LINE 23
```

The CANCEL TRACE command with the /ALL qualifier cancels all tracepoints set in the program.

```
DBG>CANCEL TRACE/ALL
DBG>GO
start at GRADES\%LINE 23

Student name? CRAIG SMYTHE
Input grade (or 0 to end input): 9
Input grade (or 0 to end input): 9
Input grade (or 0 to end input): 0

Student: CRAIG SMYTHE                    Average:   9.0

Are you done ? (Yes/No) Y
```

Because the breakpoint is still set for line 23 the debugger continues to halt execution.

```
break at GRADES\%LINE 23
DBG>GO
start at GRADES\%LINE 23
Is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>EXIT
$
```

```
  --------------------------------------------------------
      EXAMPLE 3 -- Complex debug session of GRADES
  --------------------------------------------------------

$FORTRAN/DEBUG/LIST GRADES
$LINK/DEBUG GRADES
$RUN GRADES

                VAX-11 DEBUG Version 3.0-5

%DEBUG-I-INITIAL, Language is FORTRAN, module set to 'GRADES'

DBG> SET LOG SESSION.DAT
DBG> SET OUTPUT LOG
DBG> EXAMINE DONE
GRADES\DONE(1:4):
DBG> DEPOSIT DONE='YES'
DBG> EXAMINE DONE
GRADES\DONE(1:4):  YES
DBG> SET WATCH DONE
DBG> SHOW WATCH
watchpoint at GRADES\DONE(1:4) for 4. bytes.
DBG> GO
routine start at GRADES

Student name? JOE SMITH
Input grade (or 0 to end input): 6
Input grade (or 0 to end input): 7
Input grade (or 0 to end input): 0

Student: JOE SMITH                  Average:       6.5

Are you done ? (Yes/No) N
write to GRADES\DONE(1:4) at PC 63269
            old value = YES
            new value = N

 ----------- (using TRACE while watching DONE) -------------

DBG> SET TRACE %LINE 8
DBG> SET TRACE %LINE 13
DBG> SET TRACE %LINE 19
DBG> GO
start at 63279
trace at GRADES\%LABEL 10

Student name? MARY HAGERTY
trace at GRADES\%LINE 13
Input grade (or 0 to end input): 9
Input grade (or 0 to end input): 9
Input grade (or 0 to end input): 0

Student: MARY HAGERTY                Average:       9.0
```

```
trace at GRADES\%LINE 19

Are you done ? (Yes/No) N
write to GRADES\DONE(1:4) at PC 63269
            old value = N
            new value = N

 --------- (Stop watching DONE.  Set break point) ----------

DBG> CANCEL WATCH DONE
DBG> SET BREAK %LINE 23
DBG> GO
start at 63279
break at GRADES\%LINE 23
DBG> GO
start at GRADES\%LINE 23
trace at GRADES\%LABEL 10

Student name? GERALD HORNER
trace at GRADES\%LINE 13
Input grade (or 0 to end input): 50
Input grade (or 0 to end input): 100
Input grade (or 0 to end input): 0

Student: GERALD HORNER                   Average:       75.0
trace at GRADES\%LINE 19

Are you done ? (Yes/No) N
break at GRADES\%LINE 23

 ----------- (Try to watch TOTAL. Doesn't work.) -----------

DBG> SET WATCH TOTAL
%DEBUG-W-NOSYMBOL, symbol 'TOTAL' is not in the symbol table

 ----------- (The symbol TOTAL is in subroutine.) -----------

DBG> SHOW MODULE
module name                         symbols    size
GRADES                              yes         272
COMPUTE                             no          304
total FORTRAN modules: 2.              remaining size: 56776.

 ------ (The symbol table of subroutine must be loaded) ------

DBG> SET MODULE COMPUTE
DBG> SHOW MODULE
module name                          symbols    size
GRADES                               yes         272
COMPUTE                              yes         304
total FORTRAN modules: 2.               remaining size: 56572.


 ----------- (Now we can watch TOTAL. ) ------------
```

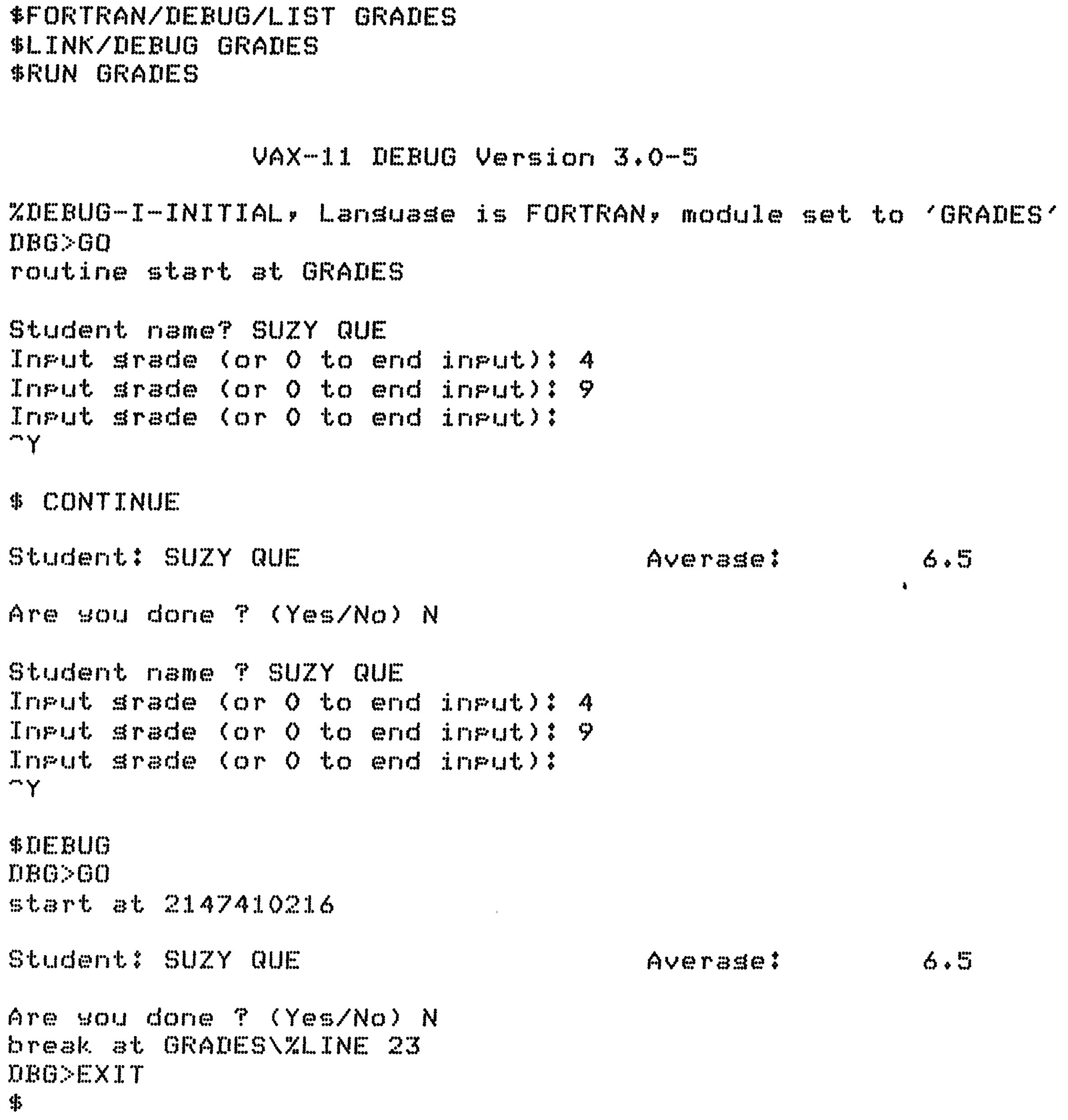
```
DBG> SET WATCH TOTAL

DBG> GO
start at GRADES\%LINE 23
trace at GRADES\%LABEL 10

Student name? JENNY GRATIN
trace at GRADES\%LINE 13

write to COMPUTE\TOTAL at PC COMPUTE\%LINE 7
            old value =    155.0000
            new value =   0.0000000E+00
DBG> GO
start at COMPUTE\%LABEL 10
Input grade (or 0 to end input): 5

write to COMPUTE\TOTAL at PC COMPUTE\%LINE 16
            old value =   0.0000000E+00
            new value =    5.000000

DBG> GO
start at COMPUTE\%LINE 17
Input grade (or 0 to end input): 6

write to COMPUTE\TOTAL at PC COMPUTE\%LINE 16
            old value =    5.000000
            new value =    11.00000

DBG> GO
start at COMPUTE\%LINE 17
Input grade (or 0 to end input): 0

Student: JENNY GRATIN                 Average:        5.5

trace at GRADES\%LINE 19

Are you done ? (Yes/No) N
break at GRADES\%LINE 23
DBG> SHOW SCOPE
scope: 0 [ = GRADES ]
DBG> SET SCOPE COMPUTE

DBG> SHOW SCOPE
scope: COMPUTE
DBG> SHOW TRACE
tracepoint at GRADES\%LINE 19
tracepoint at GRADES\%LINE 13
tracepoint at GRADES\%LINE 10

DBG> CANCEL WATCH TOTAL
```

------ (Cancel traces at lines 13 and 19, module GRADES)------

```
DBG> CANCEL TRACE %LINE 13
DBG> CANCEL TRACE %LINE 19
DBG> SHOW TRACE
tracepoint at GRADES\%LABEL 10

 ------ (Try to cancel trace at line 1Ø, module GRADES) ------

DBG> CANCEL TRACE %LABEL 10
%DEBUG-I-NOSUCHTPT, no such tracepoint
DBG> SHOW TRACE
tracepoint at GRADES\%LABEL 10

 ------ (Doesn't work because scope is not set to GRADES) ----

DBG> SET SCOPE GRADES
DBG> CANCEL TRACE %LABEL 10
DBG> SHOW TRACE
%DEBUG-I-NOTRACES, no tracepoints are set, no opcode tracing

 ----- (Add a tracepoint in main routine and subroutine ) ----

DBG> SHOW SCOPE
scope: GRADES

DBG> SET TRACE %LINE 10
DBG> SET TRACE %LINE COMPUTE\9

DBG> SHOW TRACE
tracepoint at COMPUTE\%LABEL 10
tracepoint at GRADES\%LINE 10

 ------(If duplicate labels, can specify in normal
        way if scope is set to module containing
        label to be specified.   If scope is not set,
        must specify module name also. -------------

DBG> GO
start at GRADES\%LINE 23
trace at COMPUTE\%LABEL 10
Input grade (or 0 to end input): 6
trace at COMPUTE\%LABEL 10
Input grade (or 0 to end input): 0

Student: CRAIG SMYTHE                Average:      6.0

Are you done ? (Yes/No) N
break at GRADES\%LINE 23

 -------------- (Cancel all tracepoints) -------------

DBG> CANCEL TRACE/ALL
DBG> SHOW TRACE
%DEBUG-I-NOTRACES, no tracepoints are set, no opcode tracing
DBG> GO
start at GRADES\%LINE 23
```

```
Student name? HORACE O'TOOLE

Input grade (or 0 to end input): 8
Input grade (or 0 to end input): 0

Student: HORACE O'TOOLE                   Average:     8.0

Are you done ? (Yes/No) N
break at GRADES\%LINE 23

 ---------- (Evaluate the expression TOTAL/ICOUNT) --------

DBG> SET SCOPE COMPUTE
DBG> SET BREAK %LINE 20
DBG> GO
routine start at GRADES\%LINE 23
break at COMPUTE\%LINE 20
DBG> SET WATCH COMPUTE\AVERAGE
DBG> GO
start at COMPUTE\%LINE 20
write to GRADES\AVERAGE at PC COMPUTE\%LINE 20 +7
        old value =   0.0000000E+00
        new value =    9.000000
DBG> EVALUATE TOTAL/ICOUNT
   9.000000
DBG> EXIT
$
```

EXAMPLE 4 -- Aborting and restarting the debugger



```
$FORTRAN/DEBUG/LIST GRADES
$LINK/DEBUG GRADES
$RUN GRADES


              VAX-11 DEBUG Version 3.0-5

%DEBUG-I-INITIAL, Language is FORTRAN, module set to 'GRADES'
DBG>GO
routine start at GRADES

Student name? SUZY QUE
Input grade (or 0 to end input): 4
Input grade (or 0 to end input): 9
Input grade (or 0 to end input):
^Y

$ CONTINUE

Student: SUZY QUE                    Average:        6.5

Are you done ? (Yes/No) N

Student name ? SUZY QUE
Input grade (or 0 to end input): 4
Input grade (or 0 to end input): 9
Input grade (or 0 to end input):
^Y

$DEBUG
DBG>GO
start at 2147410216

Student: SUZY QUE                    Average:        6.5

Are you done ? (Yes/No) N
break at GRADES\%LINE 23
DBG>EXIT
$
```

5.5 PROGRAM DEVELOPMENT WITH MACRO

VAX-11 MACRO is the assembly language for VAX/VMS. This language is provided with the VMS software. Examples of programs written in VAX-11 MACRO can be found in the majority of the manuals for VMS. The language reference manual and user's guide for VAX-11 MACRO are provided as part of the documentation set.

5.5.1 SOURCE FILES

A MACRO statement consists of the following fields:

- A label field. A label is a symbol used to refer to a location in your program. A label can be up to 31 characters long and can contain underscore (_) and dollar sign ($) characters. Terminate the label field with a colon (:), a double colon (::) or a space. If a label extends past column 7, place it on a line by itself; place the operator on the following line beginning at column 9. Labels are optional.

- An operator field. An operator specifies the action to be performed by a statement. The operator can be a symbol for an instruction, an assembler directive or a macro instruction. Terminate the operator field with a tab. The operator field is required.

- An operand field. The operand field contains symbolic names or specifications for the addresses of one or more operands. Operands specify instruction operands, assembler directive arguments, or macro arguments. The operand field is required.

- A comment field. A comment contains text that explains the function of the preceding statement. Ideally, you should comment every line of MACRO code, but they are optional elements of any MACRO statement. A comment can be continued from one line to the next. A comment can appear on a line by itself. Mark the beginning of any comment with a semicolon (;).

The format of a complete MACRO statement is:

```
Label: Operator Operand1,Operand2,... ;Comment
```

A statement can be continued on several lines by using a hyphen (-) as the last non-blank character before the comment field.

Conventionally, each field should begin at the column indicated by Table 5-4 for readability. The TAB key should be used the number of times indicated in the table to move the cursor to the correct column to begin input.

Table 5-4 Formatting Conventions for MACRO Statements

Field	Column	Tab Characters
Label	1	Ø
Operator	9	1
Operand	17	2
Comment	41	5

5.5.2 PREPARING THE PROGRAM FOR EXECUTION

Source programs written in VAX-11 MACRO must be assembled, not compiled. The assembler produces an object file which must be linked to produce an executable image. The following is an example of the steps of program development for a program written in MACRO:

1. $EDIT GRADES.MAR

   Creates the source file.

2. $MACRO/LIST/ENABLE=DEBUG GRADES

   Assembles the source code, producing an object file and a listing file.

3. $LINK/DEBUG GRADES

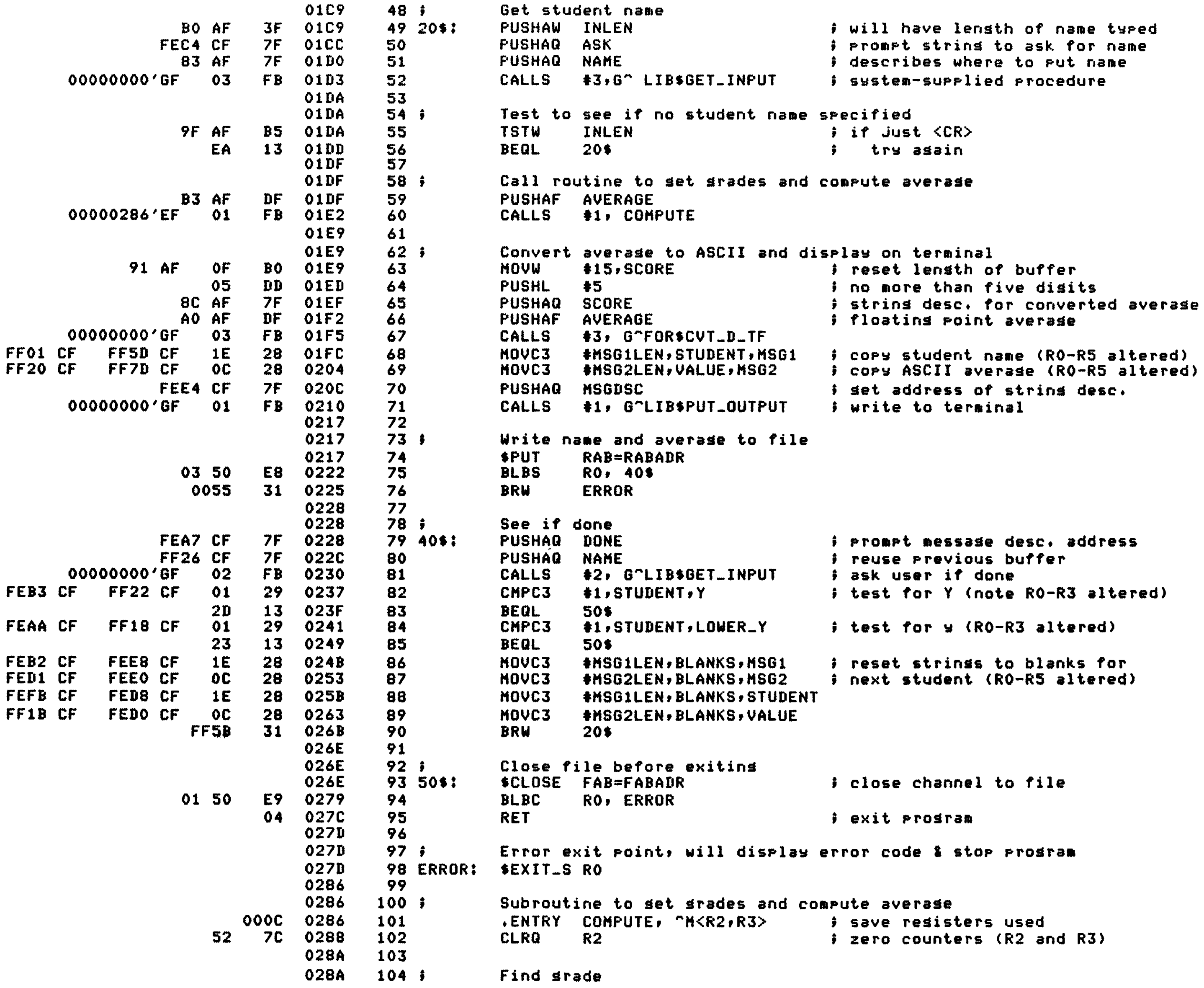
4. $RUN GRADES

5. DBG> GO

   Control passes to the symbolic debugger automatically on execution of the program. To run the program, the debug command GO should be entered.

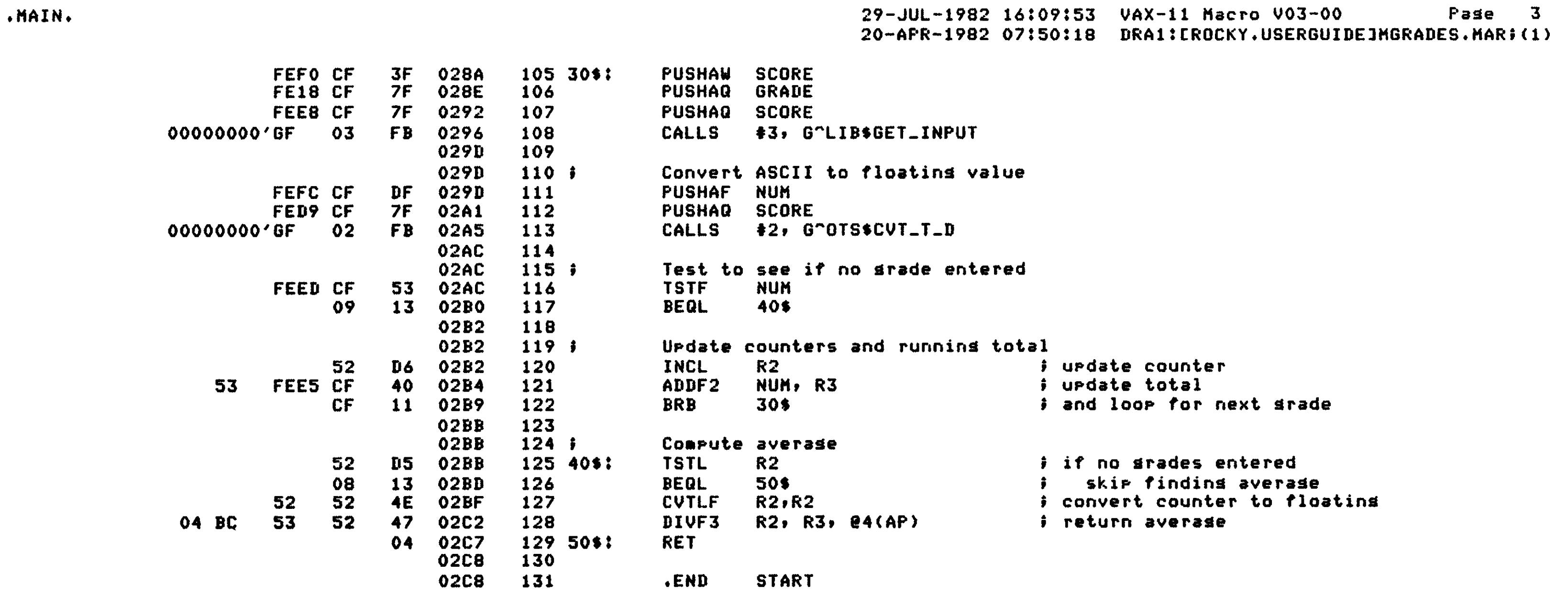
```
                                      0000     1 ;                                                   GRADES.MAR
                                      0000     2 ;
                                      0000     3 ;       RMS data file structure definitions
                                      0000     4 FABADR: $FAB    FNM=<COURSE>, FAC=<PUT>, RAT=CR
                                      0050     5 RABADR: $RAB    FAB=FABADR, RBF=MSG, RSZ=MSGSIZ
                                      0094     6 ;
                                      0094     7 ;       Messages to/from user
6E 65 64 75 74 53 0000009C'010E0000'  0094     8 ASK:    .ASCID  /Student name? /
             20 3F 65 6D 61 6E 20 74  00A2
20 74 75 70 6E 49 000000B2'010E0000'  00AA     9 GRADE:  .ASCID  /Input grade (or 0 to end input): /
20 30 20 72 6F 28 20 65 64 61 72 67   00B8
74 75 70 6E 69 20 64 6E 65 20 6F 74   00C4
                            20 3A 29  00D0
6F 79 20 65 72 41 000000DB'010E0000'  00D3    10 DONE:   .ASCID  /Are you done? (Y or N) /
6F 20 59 28 20 3F 65 6E 6F 64 20 75   00E1
                      20 29 4E 20 72  00ED
                                  59  00F2    11 Y:      .ASCII  /Y/
                                  79  00F3    12 LOWER_Y:.ASCII  /y/
                           0000003C'  00F4    13 MSGDSC: .LONG   MSGSIZ                  ; string descriptor for record
                           000000FC'  00F8    14         .ADDRESS MSG                    ; to write to file
          20 3A 74 6E 65 64 75 74 53  00FC    15 MSG:    .ASCII  /Student: /
20 20 20 20 20 20 20 20 20 20 20 20   0105    16 MSG1:   .ASCII  /                              /
20 20 20 20 20 20 20 20 20 20 20 20   0111
                   20 20 20 20 20 20  011D
                            0000001E  0123    17 MSG1LEN = . - MSG1
          20 3A 65 67 61 72 65 76 41  0123    18         .ASCII  /Average: /
20 20 20 20 20 20 20 20 20 20 20 20   012C    19 MSG2:   .ASCII  /            /
                            0000000C  0138    20 MSG2LEN = . - MSG2
                            0000003C  0138    21 MSGSIZ = . - MSG
20 20 20 20 20 20 20 20 20 20 20 20   0138    22 BLANKS: .ASCII  /                              /
20 20 20 20 20 20 20 20 20 20 20 20   0144
                   20 20 20 20 20 20  0150
                                      0156    23 ;
                                      0156    24 ;       Data storage areas
                            0000001E  0156    25 NAME:   .LONG   30                      ; string descriptor for
                           0000015E'  015A    26         .ADDRESS STUDENT                ; student name
                            0000017C  015E    27 STUDENT:.BLKB   30                      ; text of student name
                                0000  017C    28 INLEN:  .WORD   0                       ; actual length of name typed
                            0000000F  017E    29 SCORE:  .LONG   15                      ; string descriptor for
                           00000186'  0182    30         .ADDRESS VALUE                  ; grade entered
                            00000195  0186    31 VALUE:  .BLKB   15                      ; grade as ASCII string
                   00000000 00000000  0195    32 AVERAGE:.D_FLOATING 0                  ; will hold average grade
                   00000000 00000000  019D    33 NUM:    .D_FLOATING 0                  ; will hold grade in float. pt.
                                      01A5    34
                                      01A5    35 ;       Program entry point
                                003C  01A5    36         .ENTRY  START, ^M<R2,R3,R4,R5>  ; save registers used except R0,R1
                                      01A7    37
                                      01A7    38 ;       Open channel to file (and create file)
                                      01A7    39         $CREATE FAB=FABADR
                        03 50    E8   01B2    40         BLBS    R0, 10$                 ; test for errors
                         00C5    31   01B5    41         BRW     ERROR                   ; exit if error
                                      01B8    42
                                      01B8    43 ;       Connect record stream to file
                                      01B8    44 10$:    $CONNECT RAB=RABADR
                        03 50    E8   01C3    45         BLBS    R0, 20$                 ; again check for errors
                         00B4    31   01C6    46         BRW     ERROR
                                      01C9    47
```

MACRO Program Listing (Sheet 1 of 3)



```
                              01C9   ;       Get student name
                 B0 AF    3F  01C9   20$:    PUSHAW  INLEN                   ; will have length of name typed
               FEC4 CF    7F  01CC           PUSHAQ  ASK                     ; prompt string to ask for name
                 83 AF    7F  01D0           PUSHAQ  NAME                    ; describes where to put name
      00000000'GF   03    FB  01D3           CALLS   #3,G^ LIB$GET_INPUT     ; system-supplied procedure
                              01DA
                              01DA   ;       Test to see if no student name specified
                 9F AF    B5  01DA           TSTW    INLEN                   ; if just <CR>
                    EA    13  01DD           BEQL    20$                     ;   try again
                              01DF
                              01DF   ;       Call routine to get grades and compute average
                 B3 AF    DF  01DF           PUSHAF  AVERAGE
      00000286'EF   01    FB  01E2           CALLS   #1, COMPUTE
                              01E9
                              01E9   ;       Convert average to ASCII and display on terminal
            91 AF   0F    B0  01E9           MOVW    #15,SCORE               ; reset length of buffer
                    05    DD  01ED           PUSHL   #5                      ; no more than five digits
                 8C AF    7F  01EF           PUSHAQ  SCORE                   ; string desc. for converted average
                 A0 AF    DF  01F2           PUSHAF  AVERAGE                 ; floating point average
      00000000'GF   03    FB  01F5           CALLS   #3, G^FOR$CVT_D_TF
FF01 CF   FF5D CF   1E    28  01FC           MOVC3   #MSG1LEN,STUDENT,MSG1   ; copy student name (R0-R5 altered)
FF20 CF   FF7D CF   0C    28  0204           MOVC3   #MSG2LEN,VALUE,MSG2     ; copy ASCII average (R0-R5 altered)
               FEE4 CF    7F  020C           PUSHAQ  MSGDSC                  ; get address of string desc.
      00000000'GF   01    FB  0210           CALLS   #1, G^LIB$PUT_OUTPUT    ; write to terminal
                              0217
                              0217   ;       Write name and average to file
                              0217           $PUT    RAB=RABADR
                 03 50    E8  0222           BLBS    R0, 40$
                  0055    31  0225           BRW     ERROR
                              0228
                              0228   ;       See if done
               FEA7 CF    7F  0228   40$:    PUSHAQ  DONE                    ; prompt message desc. address
               FF26 CF    7F  022C           PUSHAQ  NAME                    ; reuse previous buffer
      00000000'GF   02    FB  0230           CALLS   #2, G^LIB$GET_INPUT     ; ask user if done
FEB3 CF   FF22 CF   01    29  0237           CMPC3   #1,STUDENT,Y            ; test for Y (note R0-R3 altered)
                    2D    13  023F           BEQL    50$
FEAA CF   FF18 CF   01    29  0241           CMPC3   #1,STUDENT,LOWER_Y      ; test for y (R0-R3 altered)
                    23    13  0249           BEQL    50$
FEB2 CF   FEE8 CF   1E    28  024B           MOVC3   #MSG1LEN,BLANKS,MSG1    ; reset strings to blanks for
FED1 CF   FEE0 CF   0C    28  0253           MOVC3   #MSG2LEN,BLANKS,MSG2    ; next student (R0-R5 altered)
FEFB CF   FED8 CF   1E    28  025B           MOVC3   #MSG1LEN,BLANKS,STUDENT
FF1B CF   FED0 CF   0C    28  0263           MOVC3   #MSG2LEN,BLANKS,VALUE
                  FF5B    31  026B           BRW     20$
                              026E
                              026E   ;       Close file before exiting
                              026E   50$:    $CLOSE  FAB=FABADR              ; close channel to file
                 01 50    E9  0279           BLBC    R0, ERROR
                          04  027C           RET                             ; exit program
                              027D
                              027D   ;       Error exit point, will display error code & stop program
                              027D   ERROR:  $EXIT_S R0
                              0286
                              0286   ;       Subroutine to get grades and compute average
                        000C  0286           .ENTRY  COMPUTE, ^M<R2,R3>      ; save registers used
                    52    7C  0288           CLRQ    R2                      ; zero counters (R2 and R3)
                              028A
                              028A   ;       Find grade
```

MACRO Program Listing (Sheet 2 of 3)



```
.MAIN.                                                             29-JUL-1982 16:09:53  VAX-11 Macro V03-00        Page   3
                                                                   20-APR-1982 07:50:18  DRA1:[ROCKY.USERGUIDE]MGRADES.MAR;(1)

                 FEF0 CF    3F  028A    105 30$:    PUSHAW  SCORE
                 FE18 CF    7F  028E    106         PUSHAQ  GRADE
                 FEE8 CF    7F  0292    107         PUSHAQ  SCORE
        00000000'GF    03   FB  0296    108         CALLS   #3, G^LIB$GET_INPUT
                                029D    109
                                029D    110 ;       Convert ASCII to floating value
                 FEFC CF    DF  029D    111         PUSHAF  NUM
                 FED9 CF    7F  02A1    112         PUSHAQ  SCORE
        00000000'GF    02   FB  02A5    113         CALLS   #2, G^OTS$CVT_T_D
                                02AC    114
                                02AC    115 ;       Test to see if no grade entered
                 FEED CF    53  02AC    116         TSTF    NUM
                      09    13  02B0    117         BEQL    40$
                                02B2    118
                                02B2    119 ;       Update counters and running total
                      52    D6  02B2    120         INCL    R2                      ; update counter
             53  FEE5 CF    40  02B4    121         ADDF2   NUM, R3                 ; update total
                      CF    11  02B9    122         BRB     30$                     ; and loop for next grade
                                02BB    123
                                02BB    124 ;       Compute average
                      52    D5  02BB    125 40$:    TSTL    R2                      ; if no grades entered
                      08    13  02BD    126         BEQL    50$                     ;   skip finding average
                 52   52    4E  02BF    127         CVTLF   R2,R2                   ; convert counter to floating
         04 BC   53   52    47  02C2    128         DIVF3   R2, R3, @4(AP)          ; return average
                            04  02C7    129 50$:    RET
                                02C8    130
                                02C8    131         .END    START
```

MACRO Program Listing (Sheet 3 of 3)

5.5.3 DEBUG COMMANDS

Refer to the list of debug commands in the discussion of the FORTRAN language, Section 5.6.3. Debug commands for MACRO programs can be entered using the same format except for:

1. The MACRO programmer usually has a better idea of what is occurring in their program than the high-level language programmer. Therefore, commands such as DEPOSIT and EXAMINE are used to a greater extent. Locations such as the program counter, offsets from the argument pointer, and places in memory can be examined and the information returned is usually helpful. Locations are specified using the same syntax as in a MACRO program (PC, @AP, @AP+5, R2, @AP:@AP+ØC, 4ØØ).

2. The DEFINE command is useful for making it easier to work in the debugger. Symbolic names can be assigned to addresses. After the DEFINE command is used, the symbolic name can be specified instead of the address. For example,

   ```
   DBG> EXAMINE GRADES+4
     contents of the location GRADES+4

   DBG> DEFINE PLACE = GRADES+4

   DBG> EXAMINE PLACE
     contents of the location GRADES+4
   ```

3. The DEPOSIT command can be used to change an instruction if the SET TYPE INSTRUCTION command is input first. For example;

   ```
     DBG> SET TYPE INSTRUCTION

     DBG> EXAMINE PLACE
   PLACE: MOVL   @B^A_SORTED(AP)[R2],@B^A_SORTED(AP)[R2]

     DBG> DEPOSIT PLACE='     MOVL    @B^A_ARRAY(AP)[R2],-
                                      @B^A_SORTED(AP)[R2]'

     DBG> EXAMINE PLACE
   PLACE: MOVL   @B^A_ARRAY(AP)[R2],@B^A_SORTED(AP)[R2]
   ```

4. The SET TYPE command can be used to change the default type to other types so the EXAMINE and DEPOSIT commands can be used as intended.

5.6 PROGRAM DEVELOPMENT WITH FORTRAN

5.6.1 SOURCE FILES

A line in a FORTRAN source program consists of five fields:

- Comment Indicator Field
- Statement Label Field
- Continuation Indicator Field
- Statement Field
- Sequence Number Field

There are two ways to format these fields in a FORTRAN line:

1. By means of "character-per-column" formatting

2. By means of "tab" formatting

5.6.1.1 CHARACTER-PER-COLUMN FORMATTING

Character-per-column formatting is used on VAX-11 systems to preserve compatibility with existing FORTRAN programs and those intended to be transportable between systems. The character-per-column format is the format used on punched cards, and is specified in the ANSI standard for the FORTRAN language.

The character-per-column format requires that each field of a FORTRAN line begin in a particular column. The columns that correspond to each field in a VAX-11 FORTRAN line are listed in Table 5-5.

Table 5-5 Column Conventions for FORTRAN

Column	ANSI Standard Definition
1	Comment indicator (C)
2-5	Line number
6	Continuation indicator
7-72	Valid FORTRAN statement
73-8Ø	Sequence number

5.6.1.2 TAB FORMATTING

VAX-11 FORTRAN allows the use of the TAB key to input lines of code. The compiler translates the TAB character differently depending on where it is entered in the line. Use of the TAB key makes it easier to create FORTRAN source files.

Table 5-6 shows the action taken by the compiler when it encounters the TAB character, or any characters that are not in the fields defined for them by the ANSI standard.

Table 5-6 Using Tab Formatting

If you type:	then the compiler assumes:
<TAB>text	The text is a valid FORTRAN statement and places it in columns 7-72
<TAB>#text	The number (#) is a continuation mark, and places it in column 6, followed by the text which starts in column 7
#<TAB>text	The number (#) is a line number (or statement label), and places it in columns 2-5, followed by the text which starts in column 7
C text	The entire line is a comment and ignores it
D text	The entire line is a comment and ignores it unless the /D_LINES qualifier is included with the compiler command. If the /D_LINES qualifier is included, the compiler assumes the line contains a valid FORTRAN statement, and processes it
text !comment	The text is a valid FORTRAN statement, and anything following an exclamation point is a comment; to be ignored
text in columns 73-8Ø	The text is a comment and ignores it unless the /EXTEND qualifier is included with the compiler command. If the /EXTEND qualifier is included, the compiler assumes the text is a continuation of the current line, or a valid FORTRAN statement, and processes it

5.6.2 PREPARING THE PROGRAM FOR EXECUTION

Assuming the source file has been created, the next step is compilation followed by linking; then the program can be executed. This list, an example of the steps in program development for a FORTRAN program, is followed by a partial listing of the program developed. The debugging session in section 5.4.1 uses the line numbers in the far-left column of the listing file, as well as the variable and subroutine names shown.

1. $EDIT GRADES.FOR

   Creates the source file.

2. $FORTRAN/LIST/NOOPTIMIZE/DEBUG GRADES

   Code is normally optimized by the FORTRAN compiler. Optimization involves methods which can decrease the effectiveness of the symbolic debugger. When using the debugger, the /NOOPTIMIZE qualifier should always be included to ensure close correspondence between the object code produced by the compiler and the source code.

3. $LINK/DEBUG GRADES

4. $RUN GRADES

5. DBG>GO

   Control passes to the symbolic debugger immediately on execution of the program. To run the program, the debug command GO should be entered.

Listing of Main Program

```
          PROGRAM GRADES

          CHARACTER STUDENT_NAME*30, DONE*4
          REAL AVERAGE

          OPEN (UNIT=1, FILE='Course', STATUS='New')

  10       TYPE 20
  20       FORMAT (/' Student name? ',$)
           ACCEPT 30, STUDENT_NAME
  30       FORMAT (1A30)

             CALL Get_grades_compute_average (AVERAGE)

           TYPE 40,STUDENT_NAME, AVERAGE
           WRITE (1,40) STUDENT_NAME,AVERAGE
  40       FORMAT (/' Student: ',A30,'Average: ',F10.1)

           TYPE 50
  50       FORMAT (/' Are you done ? (Yes/No) ',$)
           ACCEPT 60, DONE
  60       FORMAT (1A4)
             IF (DONE.NE.'Y' .AND. DONE.NE.'y') GOTO 10

          CLOSE (UNIT=1)
          END
```

Listing of Subroutine

```

          SUBROUTINE Get_grades_compute_average (AVERAGE)

          INTEGER ICOUNT
          REAL TOTAL, GRADE
           ICOUNT = 0
           TOTAL = 0

  10       TYPE 20
  20       FORMAT (' Input grade (or 0 to end input): ',$)
           ACCEPT 30, GRADE
  30       FORMAT (F10.0)

            IF (GRADE.NE.0) THEN
              ICOUNT = ICOUNT + 1
              TOTAL = TOTAL + GRADE
              GO TO 10
            ENDIF

  40        IF (ICOUNT.NE.0) AVERAGE = TOTAL/ICOUNT

          RETURN
          END
```

5.6.3 DEBUG COMMANDS

One example of each of the debug commands used in the sample debug session in section 5.4.1 follows:

```
SET LOG FILE.DAT

SET OUTPUT LOG

SET BREAK %LINE 23

SET MODULE COMPUTE

SET SCOPE COMPUTE

SET TRACE %LINE 8
```

(Trace and break statements can not be set at blank lines, comment lines, or lines where a FORMAT statement is specified.)

```
SET WATCH DONE

SHOW BREAK

SHOW MODULE

SHOW SCOPE

SHOW TRACE

SHOW WATCH

EXAMINE DONE

DEPOSIT DONE='YES'

EVALUATE TOTAL/ICOUNT

CANCEL BREAK %LINE 23

CANCEL TRACE %LINE 8

CANCEL TRACE/ALL

CANCEL WATCH DONE

CANCEL ALL (equivalent to CANCEL BREAK/ALL)

GO

EXIT
```

5.7 PROGRAM DEVELOPMENT WITH PASCAL

5.7.1 SOURCE FILES

All procedures and functions should be in the declaration section of a PASCAL program. Any of these may be removed and placed in a separate source file. The source file containing the main program must begin with the statement PROGRAM. The source files containing procedures and functions must begin with the statement MODULE.

5.7.2 PREPARING THE PROGRAM FOR EXECUTION

Assuming the source file has been created, the next step is compilation followed by linking; then the program can be executed. This list, an example of the steps in program development for a PASCAL program, is followed by a partial listing of the program developed. The line numbers shown in the left-hand column are used for symbolic debugger commands requiring line numbers.

1. $EDIT GRADES.PAS

   Creates the source file.

2. $PASCAL/LIST/DEBUG/NOSTANDARD/NOWARNING GRADES

   Non-standard features, including underscores (_) in identifier names, the OPEN statement, and carriage control specifications in the WRITELN statement, can be used in a VAX-11 PASCAL program. The VAX-11 PASCAL compiler displays a warning message each time it encounters one of these extensions. To suppress the messages, use the /NOSTANDARD qualifier with the compiler command. To suppress warning messages regarding unorthodox, but acceptable syntax in a program, the /NOWARNING qualifier is used.

3. $LINK/DEBUG GRADES

4. $RUN GRADES

5. DBG>GO

   Control passes to the symbolic debugger immediately on execution of the program. To run the program, the debug command GO should be entered.

Pascal Source Listing

```
PROGRAM Grades (Course, INPUT, OUTPUT);

TYPE
  Yes_no = (yes,no);
VAR
  Student_name : PACKED ARRAY [1..40] OF CHAR;
  Course : TEXT;
  Icount : INTEGER;
  Done : Yes_no;
  Grade, Total, Average : REAL;
  St, Av : PACKED ARRAY [1..10] OF CHAR;

VALUE
  St := 'Student:  ';     Av := 'Average:  ';

PROCEDURE Compute;
  BEGIN
    Icount := 0; Total := 0;
      REPEAT
       WRITE (' Input grade (or 0 to end input); ');
        READ (Grade);
         IF Grade <> 0
            THEN Icount := Icount + 1;

         Total := Total + Grade;
      UNTIL Grade = 0;
      Average := Total / Icount;
   END;

BEGIN  { Main Procedure }

 REWRITE (Course);
  REPEAT
  { Get information for one student }
          WRITELN;
          WRITE ('Student name? ');
          READ (Student_Name);
          Compute;
  { Output results to terminal and file }
          WRITELN;  WRITELN;
          WRITELN (St, Student_Name, Av, Average :3:1);
          WRITELN (Course, St, Student_Name, Av, Average :3:1);
  { Check if more students }
          WRITELN;
          WRITELN ('Are you done ? (Yes/No)  ');
          READ (Done);
  UNTIL Done = Yes;
 CLOSE (Course);

END { Program Grades }.
```

5.7.3 DEBUG COMMANDS

Refer to the list of commands in the discussion of the FORTRAN language, Section 5.6.3. Debug commands for PASCAL programs are entered using identical formats except for:

1. If the SET WATCH command is used to watch a variable that is stored on the same page in memory as a file variable, when the file variable is accessed by the program, errors occur.

2. Trace and break statements can not be set at blank lines or comment lines.

3. The COMPUTE procedure is considered as part of the main program, GRADES, so all variables are known to the debugger. Unless routines are coded in separate source files, and the MODULE statement is used, the SET SCOPE and SET MODULE commands are not useful in PASCAL.

4. The DEPOSIT command for depositing 'yes' into DONE works differently for this version of the PASCAL program, because DONE is declared as a type. The possible values of DONE are YES and NO. Since these are considered to be values, not strings, the apostrophes are not required (DBG> DEPOSIT DONE=YES). If DONE is supposed to contain a string, the DEPOSIT command would be identical.

5. Notice that the output from the debugger differs from the output when a FORTRAN module is being run.

6. The SET WATCH command can not be used with variables declared in subprograms.

5.8 PROGRAM DEVELOPMENT WITH BASIC

5.8.1 SOURCE FILES

VAX-11 BASIC can be used as though it were either an interpreter or a compiler. A fast RUN command and support for direct execution of unnumbered statements (immediate mode) gives the VAX-11 BASIC user the 'feel' of an interpreter. However, source programs created with an interactive editor can be compiled, linked, and run in the same manner as source programs written in other native-mode languages (see Section 5.1).

Table 5-7 shows the steps of program development using VAX-11 BASIC in immediate mode. More information about immediate mode or other features of VAX-11 BASIC can be found in the VAX-11 BASIC documentation. Some information is available while the user is in immediate mode, if HELP is entered at the "Ready?" prompt.

Table 5-7 BASIC Program Development Using Immediate Mode

Steps	Comments
1. $BASIC	Enters the BASIC environment.
2. Enter program	Includes line numbers
3. [LOAD file_spec]	(optional) Includes any programs needed by the main program
4. [COMPILE]	(optional) Compiles the program and any subprograms. Only used if you want to create an object file that can later that can be linked with other programs.
5. RUN	Executes program. If a CTRL/C is typed or a STOP statement is encountered, immediate mode debugging statements (see VAX-11 BASIC User's Guide) may be entered.

5.8.2 PREPARING THE PROGRAM FOR EXECUTION

Assuming the source file has been created, the next step is compilation followed by linking; then the program can be executed. This list, an example of the steps in program development for a BASIC program, is followed by a partial listing of the program developed.

1. $EDIT GRADES.BAS

2. $BASIC/LIST/DEBUG GRADES

3. $LINK/DEBUG GRADES

4. $RUN GRADES

5. DBG>GO

   Control passes to the symbolic debugger immediately on execution of the program. To run the program, the debug command GO should be entered.

Listing of Main Program

```
1 10   ! PROGRAM GRADES
1      !
1 15      OPEN 'Course' FOR OUTPUT AS FILE 1%
1      !
1 20      PRINT
2         INPUT 'Student name'; STUDENT_NAME$
1      !
1 30        CALL COMPUTE (AVERAGE)
1      !
1 40      PRINT
2         PRINT       'Student: ';          &
2                     STUDENT_NAME$,        &
2                     '                          Average: '; &
2                     AVERAGE
3         PRINT #1%
4         PRINT #1%,  'Student: ';          &
4                     STUDENT_NAME$,        &
4                     '                          Average: '; &
4                     AVERAGE
1      !
1 60      PRINT
2         INPUT 'Are you done (Yes/No)'; DONE$
4         IF DONE$ <> 'y' AND DONE$ <> 'Y' THEN GOTO 20
1      !
1 990     CLOSE 1%
1      !
1 999     END
1      !
1      !***
1      !
```

Listing of Subprogram

```
1 10000   SUB COMPUTE (AVERAGE)
1      !
1 10010   ICOUNT   = 0%
2         TOTAL = 0%
1      !
1 10020   INPUT 'Input grade (or 0 to end input)'; GRADE
1      !
1 10030   IF      GRADE <> 0%
2         THEN    ICOUNT = ICOUNT + 1%
3                 TOTAL = TOTAL + GRADE
4                 GOTO 10020
1      !
1 10040   IF      ICOUNT <>  0%
2         THEN    AVERAGE = TOTAL/ICOUNT
1      !
1 10099   SUBEND
```

5.8.3 DEBUG COMMANDS

Several kinds of variables are initialized at run-time before the code in a VAX-11 BASIC program is executed. Therefore, before examining the contents of any variables, set a breakpoint at the first statement, and input the GO command. This is true for subprograms as well. The initialization is done as you enter the subprogram. Therefore, before examining variables in a subprogram, set a breakpoint at the first statement in the subprogram and GO to that point.

Most implementations of the BASIC language require the user to input line numbers for each line of source code. Users of VAX-11 BASIC are not required to input any line numbers. However, each line must be numbered in the listing so the user can use the debugger and specify a particular line. The VAX-11 BASIC compiler does not generate more line numbers in the listing file. Instead, the line numbers in the source code are used in debug commands in a special way.

In VAX-11 BASIC, several source statements can share the same line number. The first source statement associated with a line number is assigned the number 1, which appears in the left-hand column of the listing file. The second source statement is assigned the number 2, and so on. In some cases, a particular source statement is continued over several text lines. In the listing, each line will be assigned the same number.

To designate a particular source line to the debugger, specify the line number associated with that line. If the statement is the second statement associated with the line number, specify the line number, a period, and the number 2 (line_number.2).

For example, look at the statement with the line number 4Ø in the listing of the GRADES program. Four source statements are associated with line 4Ø. To set a breakpoint at each, the following commands should be used:

```
DBG>SET BREAK %LINE 4Ø.1
DBG>SET BREAK %LINE 4Ø.2
DBG>SET BREAK %LINE 4Ø.3
DBG>SET BREAK %LINE 4Ø.4
```

The first line number can be specified using 4Ø.1 or 4Ø (the 1 is implied). Line numbers can be specified in the same manner for other debug commands requiring them.

5.9 PROGRAM DEVELOPMENT WITH COBOL

5.9.1 SOURCE FILES

The VAX-11 COBOL compiler accepts two source program coding formats: **ASNI standard** and **terminal.** Both formats are described in terms of character positions in a line. The ANSI standard, (sometimes call conventional), format is based on the traditional COBOL format as applied to an 8Ø-column punched card. The terminal format is a DEC-specified format for convenient use with an interactive text editor.

Table 5-8 compares the two formats. Notice that the terminal format does not allow the sequence number or identification fields, and both formats accept tab characters or carriage return characters as line terminators.

Table 5-8 Character Positions in COBOL Source Files

	COLUMNS	
Fields	ANSI Standard	Terminal
Sequence numbers	1-6	not used
Continuation/Comment Indicator Area	7	1
Area A	8-11	1-4
Area B	12-72	5-56
Identification Field	73-8Ø	not used

Tab stops are defined by the compiler depending on the format used.

For ANSI standard format, they are set at:

7, 8, 12, 2Ø, 28, 36, 44, 52, 6Ø, 68, 73

For terminal format they are set at:

5, 13, 21, 29, 37, 45, 53, 61, 66

Terminal format is the compiler default. The use of terminal format saves a considerable amount of space in a source file on disk as compared to the use of ANSI standard format. For this reason, if you have files on a disk which are in ANSI standard format, you may wish to convert them to terminal format using the REFORMAT utility. This utility can also be used to convert a file in terminal format to conventional format. The DCL command to invoke the REFORMAT utility is:

```
$RUN SYS$SYSTEM:REFORMAT
```

The utility will then prompt you for all pertinent information. The REFORMAT utility is described in the VAX-11 COBOL User's Guide.

5.9.2 PREPARING THE PROGRAM FOR EXECUTION

Assuming the source file has been created, the next step is compilation followed by linking; then the program can be executed. When you compile a COBOL program, the compiler will assume that you are using terminal format unless you specify the /ANSI_FORMAT qualifier.

This list, an example of the steps in program development for a COBOL program, is followed by a partial listing of the program developed. The line numbers shown in the left-hand column can be used with symbolic debugger commands requiring line numbers.

1. `$EDIT GRADES.COB`

2. `$COBOL/LIST/DEBUG GRADES`

   or

   `$COBOL/ANSI_FORMAT/LIST/DEBUG GRADES`

3. `$LINK/DEBUG GRADES`

4. `$RUN GRADES`

5. `DBG>GO`

   Control passes to the symbolic debugger immediately on execution of the program. To run the program, the debug command GO should be entered.

Source Listing

```
*  PROGRAM GRADES
*
 IDENTIFICATION DIVISION.
*
 PROGRAM-ID. GRADES.
*
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT COURSE ASSIGN TO 'COURSE'.
 DATA DIVISION.
 FILE SECTION.
 FD  COURSE
     LABEL RECORDS ARE STANDARD.
 01  OUT_REC         PIC X(72).

 WORKING-STORAGE SECTION.
 01  STUDENT_NAME    PIC X(40).
 01  AVERAGE         PIC 999V999  COMP.
 01  DONE            PIC X(4).
 01  OUT_LINE.
    05       FILLER          PIC X(9)  VALUE IS 'Student: '.
    05       OUT_NAME        PIC X(40).
    05       FILLER          PIC X(16) VALUE IS ' Average: '.
    05       OUT_AVG         PIC ZZ9.999.

 PROCEDURE DIVISION.
 BEGIN.
     OPEN OUTPUT COURSE.

 ACCEPT-STUDENT.
     DISPLAY ''.
     DISPLAY ''.
     DISPLAY 'Student name? ' WITH NO ADVANCING.
     ACCEPT STUDENT_NAME.
            CALL 'Get_grades_compute_average' USING BY REFERENCE AVERAGE.
     MOVE STUDENT_NAME TO OUT_NAME.
     MOVE AVERAGE      TO OUT_AVG.
     DISPLAY ''.
     DISPLAY ''.
     DISPLAY OUT_LINE.
     WRITE OUT_REC FROM OUT_LINE.

     DISPLAY ''.
     DISPLAY 'Are you done (Y/N)? ' WITH NO ADVANCING.
     ACCEPT DONE.
     IF DONE IS NOT EQUAL TO 'Y' AND DONE IS NOT EQUAL TO 'y'
          THEN GO TO ACCEPT-STUDENT.

     CLOSE COURSE.
     STOP RUN.
     END PROGRAM GRADES.
```

Source Listing of Subprogram

```




 IDENTIFICATION DIVISION.
*
 PROGRAM-ID. Get_grades_compute_average.
*
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  IN_GRADE           PIC 999.
 01  GRADE              PIC 999   COMP.
 01  ICOUNT             PIC 999   COMP.
 01  TOTAL              PIC 999   COMP.

 LINKAGE SECTION.
 01  AVERAGE            PIC 999V999 COMP.

 PROCEDURE DIVISION USING AVERAGE.
 BEGIN.
     MOVE ZERO TO ICOUNT.
     MOVE ZERO TO TOTAL.

         DISPLAY "".
         DISPLAY "".
         DISPLAY "(Grades must be 3-digits long.  Pad with leading 0's.)"
         DISPLAY "".
 ACCEPT-GRADE.
     DISPLAY "Enter grade (or 000 to end input): "
          WITH NO ADVANCING.

     ACCEPT IN_GRADE.

     IF IN_GRADE IS NOT EQUAL TO 0 THEN
             ADD 1 TO ICOUNT
             ADD IN_GRADE TO TOTAL
             GO TO ACCEPT-GRADE
     END-IF.

     IF ICOUNT IS NOT EQUAL TO 0 THEN
             DIVIDE ICOUNT INTO TOTAL GIVING AVERAGE.

     EXIT PROGRAM.

     END PROGRAM Get_grades_compute_average.
```

5.9.3 DEBUG COMMANDS

Debug commands for programs written in COBOL are identical to those for FORTRAN programs (see Section 5.6.3) with the exception of:

1. SET WATCH is not available when files are used.

2. STEP is used to step 1 instruction at a time. The specification of a certain number of steps is not available.

3. The TYPE command can be used to type out source statements in modules. Line numbers specified are those output in the listing file. TYPE is unique to COBOL. An example follows:

```
DBG> TYPE 1:4
module GRADES
     1: *  PROGRAM GRADES
     2: *
     3: IDENTIFICATION DIVISION.
     4: *
DBG> TYPE 55
module GRADES
    55:
DBG> TYPE 65:69
module GRADES
%DEBUG-W-NOLINXXX, lines 65:69 do not exist in module GRADES
DBG> SHOW MODULE
module name                     symbols   language   size
GRADES                          yes       COBOL       560
COMPUTE                         no        COBOL       364
COB$RMS_BLOCKS                  no        BLISS        52
LIB$AB_CVTTP_U                  no        MACRO       104
total modules: 4.         remaining size: 56980.
DBG> SET MODULE COMPUTE
DBG> TYPE 65:69
module GRADES
%DEBUG-W-NOLINXXX, lines 65:69 do not exist in module GRADES
DBG> TYPE COMPUTE\65:69
module compute
    65: 01      GRADE           PIC 999   COMP.
    66: 01      ICOUNT          PIC 999   COMP.
    67: 01      TOTAL           PIC 999   COMP.
    68:
    69: LINKAGE SECTION.
```

```
DBG> SET SCOPE COMPUTE
DBG> TYPE 65:69
module compute
    65: 01      GRADE           PIC 999   COMP.
    66: 01      ICOUNT          PIC 999   COMP.
    67: 01      TOTAL           PIC 999   COMP.
    68:
    69: LINKAGE SECTION.
DBG> EXIT
```

CHAPTER 6

SIMPLIFYING A USER SESSION

User sessions can be simplified through the use of command procedures and symbols. This is especially helpful for the frequent user. Command procedures are usually created to perform specified or repetitive jobs.

A command procedure is a text file, created by an editor. It contains a list of DCL commands, and is formatted in a standard way. The DCL interpreter can read the DCL commands from the file instead of from the user's terminal. By placing commonly used DCL command sequences in a file, the user can more easily interact with the system.

A symbol is a series of characters representing part or all of a DCL command. The series of characters for a symbol is chosen by the user. Using symbols gives the user the ability to tailor the DCL command language for themselves.

6.1 CREATING A COMMAND PROCEDURE

Command procedures are an easy way of entering commonly-used DCL command sequences. Since all the necessary commands are in a file, the exact order and form of the commands is recorded. Once entered into a file, a command procedure can be used as many times as needed. The continued reuse of a command procedure saves users the time needed to find the correct command sequence and enter it each time.

When the user is working at a terminal, the DCL interpreter outputs a prompt, $, to indicate when it is ready to receive a command. When a command procedure is created, each command listed should be preceded by a $. Any line not preceded by a $ will be treated as data, not as a command. (Note: The $ should always be entered in the first column of the line.)

After you create a command procedure, you can execute all the commands in it with a single command. For example, suppose a procedure named PROCESS.COM contains the lines:

```
$FORTRAN/LIST PROGRAM.FOR
$PRINT PROGRAM.LIS
$LINK PROGRAM.FOR
$RUN PROGRAM.FOR
```

The commands in this file can be executed by entering the following command at the DCL prompt:

```
$@PROCESS
```

The @ (execute procedure) command assumes the filetype is .COM. Each command in the command procedure is executed in the order specified.

The commands in a DCL command procedure are not normally displayed as they are executed. The user will see any output or error messages normally associated with the command, but not the command itself. If the user inputs the SET VERIFY command, the commands will be seen. Commands will continue to be seen for all command procedures subsequently executed until a SET NOVERIFY is input.

The SET commands can be included in the command procedure or entered interactively by the user. Users will find the SET VERIFY command to be especially helpful when a new procedure has been created, and they are trying to determine whether it is working as intended or not. For example:

User creates three files:

```
  SHOW.COM             SHOW2.COM               DO.COM

 $SHOW TIME           $SET VERIFY             $@SHOW
 $SHOW USERS          $SHOW TIME              $@SHOW2
                      $SHOW USERS             $@SHOW
```

User invokes the DO.COM procedure in an interactive session and observes the output:

```
$@DO
   5-APR-1982 09:57:25
VAX/VMS Interactive Users - Total = 1
      5-APR-1982 09:57:25.54

    TTA1:     DRAGI           00040035

$SHOW TIME
   5-APR-1982 09:57:25
$SHOW USERS
VAX/VMS Interactive Users - Total = 1
      5-APR-1982 09:57:26.07

    TTA1:     DRAGI           00040035

$@SHOW
$SHOW TIME
   5-APR-1982 09:57:26
$SHOW USERS
VAX/VMS Interactive Users - Total = 1
      5-APR-1982 09:57:26.95

    TTA1:     DRAGI           00040035
```

Commands (if SET VERIFY is activated), output from commands, and error messages can be saved in a file by including the /OUTPUT qualifier to the @ command:

```
$@DO/OUTPUT=DO.LIS
$
```

Errors will appear on the terminal as well as in the output file. If no errors occur, no output will be seen. The output file must be printed or typed to observe the results.

Commands should not be abbreviated in a command procedure. Using the complete command makes the procedure more readable and self-commenting. If extra comments are needed within a procedure, they can be placed anywhere in a line if preceded by an exclamation point (!). The DCL interpreter ignores everything on a line after an ! is read. Therefore, comments are not executed.

DCL commands are normally executed in the order they appear in the command procedure, in the same way statements are executed in the order they appear in programs.

Some DCL commands are available to change the order of execution, including IF, GOTO, and EXIT. Other commands are available for the manipulation of files from a command procedure, including OPEN, READ, and WRITE. These commands are not needed in simple procedures, but more sophisticated users can learn about them through the use of the HELP facility.

If any command executed in a command procedure causes an error or severe error to occur, an appropriate message will be output and the command procedure will be terminated. Successful commands, and those causing warning messages to be output, will not terminate the procedure.

6.1.1 THE LOGIN.COM PROCEDURE

Most users will create at least one command procedure with the name LOGIN.COM. This procedure, stored in the user's top-level directory, is executed by the system each time the user logs in. The LOGIN.COM file typically contains commands to change the user environment, output information to the user, and create symbols (see Section 6.2). For example:

The LOGIN.COM file contains the following lines:

```
$SET VERIFY
$!
$! Obtain information
$!
$SHOW SYSTEM
$SHOW USERS
$SHOW PROCESS
$SHOW TIME
$!
$! Modify the environment
$!
$SET TERMINAL/VT52
$!
$! Create symbols
$!
```

After the LOGIN.COM has been created or modified, the user should always test it before logging out:

```
$@LOGIN
```

This precaution is necessary, because if the procedure contains certain errors, the user may not be able to log back in again. When the procedure executes without error, the user can log out and log in to observe that the system executes the procedure automatically.

6.2 CREATING SYMBOLS

Symbols can be used to create synonyms for DCL commands or parts of DCL commands. Symbols are created through the use of an **assignment statement.** For example, the symbol LIST could be defined to equate to the DCL command DIRECTORY as follows:

```
$LIST == "DIRECTORY"
```

The symbol can be used as follows:

```
$LIST

(output for the directory command is seen)
```

When the user inputs LIST as a command, the DCL interpreter looks in the table where symbols are stored and translates LIST to be DIRECTORY. Then the interpreter executes the DIRECTORY command.

A symbol can also be equated to a portion of a command, as follows:

```
$FL == "FORTRAN/LIST"
```

Since FORTRAN/LIST requires a file specification to be complete, FL also requires a file specification:

```
$FL PRGM.FOR
```

A symbol created by a user is valid only for that user. Two kinds of symbols can be created, local and global. Global symbols are most useful to the average user, and that is the kind of symbol created in the previous examples. To list all global symbols created during a user session, the command "SHOW SYMBOL/GLOBAL/ALL" can be entered. To delete a symbol, the command "DELETE/SYMBOL/GLOBAL symbol_name" can be entered.

6.2.1 PARAMETER SYMBOLS

Eight local symbols, called parameter symbols, are created automatically for the user whenever a command procedure is invoked. These can be used to input information to the procedure at the time of activation. The names of these symbols are P1, P2, P3, P4, P5, P6, P7, and P8.

Procedures are executed with the command:

```
$@file_specification
```

Information can be input optionally on this command line following the file specification. The information can be any string of characters. The first piece of information is automatically equated to the P1 symbol. The second piece of information is equated to the P2 symbol, and so on, up to eight pieces of information.

Parameter symbols exist for the duration of the command procedure only. When the command procedure is done, the symbols are deleted by the system. If the procedure is invoked again, the symbols are re-created.

Parameter symbols are commonly used to input file names or instructions to the procedure, as shown in the following examples:

Example 6-1

The file PROCESS.COM contains the following statements:

```
$SET VERIFY
$FORTRAN/LIST 'P1'    !Notice that P1 is enclosed
$PRINT 'P1'           !in apostrophes to indicate
$LINK 'P1'            !to the DCL interpreter that
$RUN 'P1'             !it is a symbol
```

A user executes the procedure, passing PRGM as the value of P1:

```
$
$@PROCESS PRGM
$FORTRAN/LIST PRGM
$PRINT PRGM
  Job 5Ø9 entered on queue SYS$PRINT
$LINK PRGM
$RUN PRGM
HI
```

During the execution of the command procedure, the DCL interpreter will substitute PRGM wherever P1 appears. Default values for portions of file specifications not input are available within command procedures. Therefore, PRGM is sufficient, as the FORTRAN compiler will add the file_type of .FOR, the PRINT program will add the file_type of .LIS, the LINK program will add the file_type of .OBJ, and the RUN program will add the file_type of .EXE.

Example 6-2

The file LOGOUT.COM contains the following lines:

```
$SET VERIFY
$!
$IF P1.NES."PURGE" THEN GOTO LOGOUT    !String
$!                                      comparison
$PURGE [...]*.*
$!
$LOGOUT:          !Label - indicated by colon
$!                         terminator
$LOGOUT
```

When the user inputs the command @LOGOUT, with no parameter, the LOGOUT command will be executed. If the user inputs any string as a parameter other than PURGE, the LOGOUT command will be executed. If the PURGE string is input for P1, then the PURGE command will be executed followed by the LOGOUT command.

```
$@LOGOUT
   user is logged out

$@LOGOUT JKLM
   user is logged out

$@LOGOUT PURGE
   all files are purged
   user is logged out
```

6.2.2 INTERPRETATION OF SYMBOLS

Symbols can be used in several places. The previous examples have shown three ways a symbol can be used. Many rules exist by which DCL interprets symbols. Some of these rules follow:

1. The DCL interpreter assumes any string input after the $ prompt, during an interactive session, could be a symbol. Therefore, the interpreter always checks the symbol table to see if the first string input is a symbol.

2. In some DCL commands used within command procedures, such as IF, WRITE, and INQUIRE, the interpreter assumes certain strings could be symbols. If the string is found in the symbol table, the substitution is made and the command is executed.

3. In other DCL commands, the interpreter must be informed that a string is a symbol, such as TYPE, PRINT, FORTRAN, and LINK. The interpreter can be informed in these cases by enclosing the symbol in single apostrophes.

   For example, in the case of the FORTRAN command, if the user inputs FORTRAN P1, the FORTRAN program will add the file_type .FOR to P1, and attempt to compile P1.FOR.

   To inform the interpreter that P1 is really a symbol equated to a value (in this case, the name of the file), P1 should be enclosed in quotes. The command input should be FORTRAN 'P1'. The interpreter substitutes the value equated to P1; the FORTRAN program adds the file_type of .FOR, and the correct file is compiled.

For most DCL commands, the DCL interpreter must be informed (by using apostrophes) that a string is a symbol. If the documentation states that the input string may be a symbol, then no apostrophes are needed. For example, look at the documentation for IF and WRITE, using the HELP command as follows:

```
$HELP IF parameters

$HELP WRITE parameters
```

Contrast that documentation with the information output for the FORTRAN command:

```
$HELP FORTRAN parameters
```

CHAPTER 7

PRODUCING FORMATTED TEXT OUTPUT

The RUNOFF utility is a text formatter. The utility accepts an input file and produces an output file. The input file contains text and RUNOFF formatting commands. The output file contains the formatted text. The formatted output file includes line feeds and form feeds at appropriate points for output on a line printer. By learning and using a few RUNOFF commands, users can produce professional looking text.

7.1 USING RUNOFF

To use the RUNOFF utility, the following steps should be taken:

1. Create the input file using an editor

   - file_type should be .RNO

2. Exit the editor, saving the contents

3. Create the formatted output file by using the command:

   ```
   $RUNOFF file_name
   ```

4. Print or type the output file

   - file_type is .MEM

While the RUNOFF utility is processing the input file, it may encounter incorrect commands. If this occurs, an error message will be output describing the error. The message usually includes the number of the line in the input file where the error occurred. To correct the error, the input file must be modified using an editor. After modification, the new version of the input file should be processed by RUNOFF to produce a new output file.

When the input file has been processed successfully, the output file should be examined. If the output is formatted as intended, a final copy can be printed. Otherwise, the input file should be modified with an editor to reflect any corrections, and the new version of the input file should be processed by RUNOFF. These steps should be repeated until the output file is acceptable.

Several qualifiers are available to be used with the RUNOFF command. Use the DCL HELP utility to learn more about these qualifiers.

7.2 INPUT FILES

RUNOFF commands always begin with a period (.). This period must appear in the first column followed by the command (no space between period and first word of command).

Some RUNOFF commands are normally included before or after text. Others are usually included at the beginning of the input file rather than repetitively within the file. Some special commands called symbols appear within text strings. Most commands are input in one of the following formats:

1. .command

2. .command number

3. .command;TEXT

4. .command
   TEXT

5. .command;TEXTsymbolTEXTsymbolTEXT

The next section of this chapter contains examples of input files and their corresponding output files. Tables listing all commands used can be found in the section following the example listings. A few commands are discussed further within the examples. Some of the output files included several form feeds to display the action of RUNOFF commands. These form feeds will be indicated by the following (which would not normally be seen in an output file):

```
----------- <Form Feed> ------------
```

```
.page size 58,7Ø
.title EXAMPLE 1
.first title
.autoparagraph
.set paragraph Ø,1,2
.flags bold
.center;Introduction

.blank 2

 This is example number one.  This paragraph will be
automatically formatted by RUNOFF so all lines will
look like they are the same length.  Notice that the
.autoparagraph command is set at the beginning of the
file.  Since this paragraph begins with a space, it
will be formatted as a new paragraph.

This is a new paragraph.  RUNOFF starts a new paragraph
if a blank line or a space at the beginning of a line
are read.

 All paragraphs are output with the .set paragraph
format.  Therefore;

.list "o"
.le;Paragraphs are not indented
.le;One blank line is output before each paragraph
.le;If only one line of a paragraph can fit on a page,
a form feed is done first.
.end list

 To make the input file easy to read, paragraphs
should be separated by a blank line ^*and\* begun with
a space.  (If this file were processed by RUNOFF, and
the resulting file were printed, the 'and' in the
previous sentence would be bolded.)

 The other commands listed at the beginning of this
file are:

.list
.le;_.title
.le;_.first title
.le;_.flags bold
.le;_.center
.le;_.page size
.end list
```

```
EXAMPLE 1                                                      Page 1


                             Introduction



This is example number one.   This  paragraph  will  be  automatically
formatted  by  RUNOFF  so  all  lines will look like they are the same
length.   Notice  that  the  .autoparagraph  command  is  set  at  the
beginning  of  the file.  Since this paragraph begins with a space, it
will be formatted as a new paragraph.

This is a new paragraph.  RUNOFF starts a new  paragraph  if  a  blank
line or a space at the beginning of a line are read.

All paragraphs are output with the .set paragraph format.  Therefore;

      o  Paragraphs are not indented

      o  One blank line is output before each paragraph

      o  If only one line of a paragraph can fit on  a  page,  a  form
         feed is done first.


To make the input file easy to read, paragraphs should be separated by
a  blank  line and begun with a space.  (If  this file were  processed
by   RUNOFF,  and  the  resulting  file  were  printed,  the  'and' in
the previous sentence would be bolded.)


The other commands listed at the beginning of this file are:

     1.  .title

     2.  .first title

     3.  .flags bold

     4.  .center

     5.  .page size
```

```
.page size 20,70        !Range of values is: length, 13 - 9999
.!                                           width, 3 - 150
.title EXAMPLE 2
.set paragraph 5,1,2
.spacing 2              !Causes all lines in the output file
.!                       to be double-spaced

.center;Introduction

.paragraph
This paragraph begins with an indentation of 5 spaces, as
specified in the .set paragraph command.  The title, EXAMPLE 2,
should not appear until the second page of this document.
When 20 lines have been entered on this page, the RUNOFF
formatter will automatically insert a form feed into the output
file, and begin a new page.

.spacing 1 !Changes the spacing to single spacing again
.!         Notice that comments do not appear in the output file

.header level 1 Discussion of header levels and paragraphs

.paragraph
Notice that the title, 'DISCUSSION OF HEADER LEVELS AND
PARAGRAPHS', follows the number 1.0 in the output file.
A new section of text, usually discussing the item described
in the title, begins after the section header.  Sections
are set apart by blank lines before and after the number and
name of the sections. (Notice that this paragraph is also
indented by 5 spaces, and that it is necessary to use the
.paragraph command to indicate a new paragraph.)  If the
.paragraph command is not used, and .autoparagraph is not
set, all text will be included in the same paragraph.
Notice that if .autoparagraph was set, this paragraph would
be set apart as a separate paragraph.  Since it is not set,
this paragraph is included as part of the preceding paragraph.

.header level 2 More discussion of paragraphs

.paragraph 3,1,2
 The .paragraph command can be used to change the indentation
and other characteristics of paragraphs also.
.paragraph
 Notice that all letters in the top header level are
capitalized by default, and the first letters of the second
level are capitalized.

.header level 3 displaying header level 1 characteristics

.paragraph
 For third level header levels, the first character of each
word in the title is capitalized, and the title is followed
by a hyphen.
```

```
                          Introduction

     This  paragraph  begins  with  an  indentation  of  5  spaces,  as

specified in the .set paragraph command.  The title, EXAMPLE 2, should

not appear until the second page of this document.  When 2Ø lines have

been  entered  on  this  page, the RUNOFF formatter will automatically

insert a form feed into the output file, and begin a new page.

                 ----------- <Form Feed> ------------

EXAMPLE 2                                                        Page 2


1.Ø  DISCUSSION OF HEADER LEVELS AND PARAGRAPHS

     Notice   that   the   title,  'DISCUSSION  OF  HEADER   LEVELS   AND
PARAGRAPHS', follows the number 1.Ø in the output file.  A new section
of text, usually discussing the item described in  the  title,  begins
after  the  section  header.  Sections  are  set apart by blank lines
before and after the number and name of the  sections.   (Notice  that
this  paragraph is also indented by 5 spaces, and that it is necessary
to use the .paragraph command to indicate a  new  paragraph.)  If  the
.paragraph  command  is  not  used, and .autoparagraph is not set, all
text  will  be  included  in  the  same  paragraph.   Notice  that  if
.autoparagraph  was  set,  this  paragraph  would  be  set  apart as a
separate paragraph.  Since it is not set, this paragraph  is  included
as part of the preceding paragraph.

                 ----------- <Form Feed> ------------

EXAMPLE 2                                                        Page 3


1.1  More Discussion Of Paragraphs

   The .paragraph command can be used to change  the  indentation  and
other characteristics of paragraphs also.

   Notice that all letters in the top header level are capitalized  by
default, and the first letters of the second level are capitalized.


1.1.1  Displaying Header Level 1 Characteristics -

   For third level header levels, the first character of each word  in
the title is capitalized, and the title is followed by a hyphen.
```

```
.page size 58,70
.title EXAMPLE 3
.first title
.autoparagraph
.set paragraph 0,1,2
.center;INTRODUCTION
.blank 2

 In many types of documents, reports, and memos, lists of items
must be created.  When creating a list, items are usually set
apart by numbering, or bulleting each item.  These methods are
shown in Example#1.  This example shows other methods of
identifying list elements by using the .display element command.
The colors of the United States of America's flag are:

.blank 2
.indent 2;Lowercase letters:
.list
.display element " ",LL," "
.le;red
.le;white
.le;blue
.end list

.blank 2
.indent 2;Lowercase letters followed by a period:
.list
.display element " ",LL,"."
.le;red
.le;white
.le;blue
.end list

.blank 2
.indent 2;Uppercase letters surrounded by parentheses:
.list
.display element "(",LU,")"
.le;red
.le;white
.le;blue
.end list

.blank 2
.indent 2;Lowercase Roman numerals:
.list
.display element " ",RL,"."
.le;red
.le;white
.le;blue
.end list
```

```
EXAMPLE 3                                                        Page 1

                              INTRODUCTION


In many types of documents, reports, and memos, lists of items must be
created.  When  creating  a  list,  items  are  usually  set apart by
numbering, or  bulleting  each  item.  These  methods  are  shown  in
Example 1.  This  example  shows  other  methods  of identifying list
elements by using the .display element command.  The  colors  of  the
United States of America's flag are:


  Lowercase letters:

     a   red

     b   white

     c   blue


  Lowercase letters followed by a period:

     a.  red

     b.  white

     c.  blue


  Uppercase letters surrounded by parentheses:

    (A)  red

    (B)  white

    (C)  blue


  Lowercase Roman numerals:

     i.  red

    ii.  white

   iii.  blue
```

```
.page size 58,7Ø
.title EXAMPLE 4
.first title
.subtitle
.autosubtitle
.autoparagraph
.set paragraph 2,1,2
.center;INTRODUCTION

 Several commands are used to center, set apart, or display
text in unconventional manners.  The commands include:
.blank 1
.list Ø,"-"
.le;_.literal
.le;_.end literal
.le;_.note
.le;_.end note
.le;_.right margin _#
.le;_.left margin _#
.end list

.note
The notes command is used to set text apart from other
text by indenting the text an equal distance from each
margin. The word NOTE is placed before the indented text.
.end note

.left margin +2
.right margin -2
.blank 1
.center;NOTE
 To create a note that appears differently from the normal
NOTE command's output, the .left margin and right margin
commands can be used.  These commands reset the margin,
and all text is then formatted within the new margins.
The margins can be reset to the original margins at any
time, or they can be reset to other new margins.
.right margin +2
.left margin -2

.blank 1
.literal
                Some text is required to appear
in a certain format regardless of what the margins are.
For this purpose, the .literal command
is used.  Text following the .literal
command appears in the output file to be identical to
the text in the input file until a .end literal command
is reached.
.blank 2
Notice that commands are ignored within a literal.
 .end literal
```

```
EXAMPLE 4                                                       Page 1


                             INTRODUCTION

  Several commands are used to center, set apart, or display  text  in
unconventional manners.  The commands include:

      -  .literal
      -  .end literal
      -  .note
      -  .end note
      -  .right margin #
      -  .left margin #


                                 NOTE

               The notes command is used  to  set  text
               apart  from  other text by indenting the
               text an equal distance from each margin.
               The  word  NOTE  is  placed  before  the
               indented text.


                                 NOTE

    To create a note that appears differently from the  normal  NOTE
  command's  output,  the .left margin and right margin commands can
  be used.  These commands reset the margin, and all  text  is  then
  formatted within the new margins.  The margins can be reset to the
  original margins at any time, or they can be reset  to  other  new
  margins.

                Some text is required to appear
in a certain format regardless of what the margins are.
For this purpose, the .literal command
is used.  Text following the .literal
command appears in the output file to be identical to
the text in the input file until a .end literal command
is reached.
.blank 2
Notice that commands are ignored within a literal.
```

```
.require "FORMAT.RNO"
.title EXAMPLE 5
.number chapter 5

.!
.!  The FORMAT.RNO file is listed below.  It contains the
.! general formatting information used by this example:
.!
.!  Contents of FORMAT.RNO
.!
.! .page size 58,70
.! .first title
.! .subtitle
.! .autosubtitle
.! .autoparagraph
.! .set paragraph 2,1,2
.! .center;INTRODUCTION
.!

.header level 1 Chapters

  The output of
the .chapter command can be seen by looking at the beginning
of each chapter.  The .number chapter n command was used at
the beginning of each chapter to indicate the chapter number.
The new number was then incorporated as part of the page
identification.

.page

 Notice that a form feed is done even though 58 lines have not
been output because of the .page command.

.header level 2 Layout
 The default was used for the .layout command, and all pages
are numbered using decimal numbers.  Notice that the first
header level is used as the subtitle.


.page
.autosubtitle 2
.header level 2 Commands not shown in examples

 Notice that the second header level title is used as the
subtitle because of the .autosubtitle command.

 These examples have contained
most of the commands listed in the following tables.  The
commands not depicted are more advanced.  Users should be
able to read the syntax of the command from the table to
incorporate it in their input file.
```

```
                           INTRODUCTION



1.Ø  CHAPTERS

  The output of the .chapter command can be seen  by  looking  at  the
beginning  of each chapter.  The .number chapter n command was used at
the beginning of each chapter to indicate the chapter number.  The new
number was then incorporated as part of the page identification.

                 ----------- <Form Feed> ------------

EXAMPLE 5                                                   Page 5-2
CHAPTERS

  Notice that a form feed is done even though 58 lines have  not  been
output because of the .page command.


1.1  Layout

  The default was used for the .layout  command,  and  all  pages  are
numbered using decimal numbers.  Notice that the first header level is
used as the subtitle.

                 ----------- <Form Feed> ------------

EXAMPLE 5                                                   Page 5-3
Commands Not Shown In Examples


1.2  Commands Not Shown In Examples

  Notice that the second header level title is used  as  the  subtitle
because of the .autosubtitle command.

  These examples have contained most of the  commands  listed  in  the
following tables.  The commands not depicted are more advanced.  Users
should be able to read the syntax of the command  from  the  table  to
incorporate it in their input file.
```

7.3 SUMMARY OF RUNOFF COMMANDS

Commonly used RUNOFF commands are summarized in several tables in the next section. Commands are listed in tables by function for reference purposes.

The tables contain commands affecting the following:

- Table 7-1 - Paragraph format
- Table 7-2 - Text format
- Table 7-3 - Creation of lists
- Table 7-4 - Symbols
- Table 7-5 - Recognition of symbols
- Table 7-6 - Title information
- Table 7-7 - Amount of text on a page
- Table 7-8 - Page identification
- Table 7-9 - General format

NOTE

Any of the commands listed in the tables can be included anywhere in the input file. Abbreviations for each command are included in parentheses under each command although command files are more readable and self-documented when abbreviations are not used. Optional portions of commands are enclosed in square brackets (e.g. [optional])

Table 7-1 Commands affecting paragraph format

Command	Effect on output file
.autoparagraph (.ap)	Enables the automatic recognition of paragraphs. A new paragraph is begun in the output file if a blank line or a line beginning with a space is read in the input file
.no autoparagraph (.nap)	Disables automatic paragraph recognition (default)
.set paragraph [i,v,t] (.spr)	Describes the format of each paragraph. i designates how many spaces to indent before text begins. v designates the number of vertical line feeds before a paragraph. t designates how many lines can be output before a form feed must be done. If the specified number of lines can not be output, the form feed is output first; then the paragraph. Default is 5,1,2.
.paragraph [i,v,t] (.p)	Specifies that the following text is a new paragraph. Needed only if .autoparagraph is not specified. Can reset paragraph characteristics (see .set paragraph)

Table 7-2 Commands used to format specific portions of text

Command	Effect on output file
.center;text (.c)	Center the specified text. Text may follow .center; or may be input on the subsequent line. Only one line will be centered.
.indent n (.i)	Indent next line n spaces to the right or left (if n is negative) of the left margin.
.left margin n (.lm)	Set the left margin to column n or; Move the left margin: - to the right if n is positive - to the left if n is negative
.right margin n (.rm)	Set the right margin to column n or; Move the right margin: - to the right if n is positive - to the left if n is negative
.break (.br or .)	End the current line without filling or justifying it and begin new line.
.literal (.lt)	Specify that the subsequent text is not to be formatted. It will appear in the output file exactly as it appears in the input file. (Caution: The TAB is the exception. TAB is is translated as a space. Use the SPACE bar instead of TAB.)
.end literal (.el)	Causes RUNOFF to begin formatting text again.
.note [title] .end note (.n and .en)	Indent text between .note and .end note from both margins. Precedes and follows the text with blank lines. Also precedes the text with the word NOTE (or optional title centered on a line.

Table 7-3 Commands used for the creation of lists

Command	Effect on output file
.list n,"c" (.ls)	Begin a list with n blank lines between each item. Each item begins with the character indicated by "c", by default, decimal numbers incremented by 1. Typical values for "c" are "o", or " ", or "-".
.display element "a","b","c" (.dle)	Identify list items. "a" and "c" are single characters to be displayed before and/or after "b" "b" is defined using a code chosen from the following list: Code Output D Decimal numbers RU Roman uppercase numerals RL Roman lowercase numerals RM Roman mixed case numerals LU Letters, uppercase LL Letters, lowercase LM Letters, mixed case (mixed case - 1st letter only is uppercase)
.list element;text (.le)	Specifies item to be listed. This command must precede each item.
.end list (.els)	Identifies the end of the list.

Table 7-4 Symbols used within text lines to format text

Symbol	Effect on output file
The following symbols are automatically enabled:	
_	Underscore. Causes any character following it to be accepted as normal text. Useful when a special symbol is to be included in text as text.
#	Number sign. Outputs exactly one space.
&	Ampersand. Underlines the character immediately following it.
^&text\&	The text between the up-arrow ampersand and backslash ampersand symbols is underlined.
The following symbols must be enabled to have any effect:	
*	Asterisk. Causes the character immediately following it to be bolded. (Appears darker if output file is printed)
^*text*	The text between the up-arrow asterisk and backslash ampersand symbols is bolded.
%	Percent sign. When inserted between two characters, causes the preceding character to be overstruck by the subsequent character.

Table 7-5 Commands used to enable/disable recognition of symbols

Command	Effect on output file
.flags bold (.fl bold)	Enables recognition of * as the bolding command.
.no flags bold (.nfl bold)	Disables recognition of * as bolding command
.flags overstrike (.fl overstrike)	Enables recognition of % as the overstrike command.
.no flags overstrike (.nfl overstrike)	Disables recognition of % as the overstrike command.

Table 7-6 Commands affecting titles output on pages

Command	Effect on output file
.title text (.t)	Includes the specified title, TEXT, as the first line on each page except the first page
.subtitle text (.st)	Enables automatic subtitling. If a subtitle, TEXT, is specified, includes it under the title on every page except the first page. If .autosubtitle is also input as a command, includes header level titles as subtitles instead
.first title (.ft)	Causes the title and subtitle to be output on the first page also
.autosubtitle n (.ast)	Causes header level titles up to and including the level indicated by n (default is 1) to be used as subtitles if .subtitle is also input as a command.
.noautosubtitle (.nast)	Disables autosubtitling
.header level n text (.hl)	Allows use of section numbering: Value of n: Type of Output 1 2.1 text 2.2 text 2 2.1.1 text 2.1.2 text 2.1.3 text 3 2.1.1.1 text 2.1.1.2 text 2.1.1.3 text
.display level code	Displays header level numbers in format according to code (see .display element for codes). Default is decimal numbers.

Table 7-7 Commands affecting amount of text on a page

Command	Effect on output file
.page size l,w (.ps)	Determines the size of each page. l designates the length (lines per page), and w designates the width (characters per line). Default is 58,60.
.page (.pg)	Starts a new page
.spacing n (.sp)	Establishes spacing between lines (l=single space, 2=double space, etc. up to 5)
.test page n (.tp)	Start a new page if there are less than n lines left on current page
.blank n (.b)	Output n blank lines

Table 7-8 Commands affecting Page Identification

Command	Effect on output file
.no number (.nnm)	Disable listing (but not counting) of page numbers.
.number page n (.nmpg)	Resume sequential page numbering, using page number n as first page. If n is not specified, use current page number.
.display number code (.dnm)	Display page numbers in format according to code (see .display element for code). Default is decimal numbers.
.chapter [title] (.ch)	Start a new chapter on a new page using title specified.
.number chapter n (.nmch)	Specify the number of the current chapter. If n is not specified, 1 is used.
.display chapter code (.dch)	Display chapter numbers in format according to code (see .display element for code). Default is decimal numbers.

Table 7-9 Commands affecting overall format of output file

Command	Effect on output file
.require "file" (.req)	Causes the specified file to be read and processed. The file usually contains commands to set up the general format of the output file.
.layout n,[m] (.lo)	Specifies the location of the title/subtitle and page identification. Use one of the following codes for n (Ø is default): Ø Title/subtitle flush left Page id flush right 1 Title/subtitle centered at top of page Page id centered at bottom of page 2 Title/subtitle flush right (odd page) and flush left (even page) Page id centered at bottom of page 3 Title/subtitle flush left Page id flush right and page numbers incremented by 1 centered at bottom of each page. (e.g. at top, page number is 4-7; at bottom, page number is 132) The second argument, m, is used to indicate the number of blank lines which should be inserted between the page id at the bottom of the page, and the last line of text. (required for codes 1 to 3)
.! comment	To include comments which will not appear in the formatted output file

CHAPTER 8

MISCELLANEOUS VAX/VMS UTILITIES

The MAIL and PHONE utilities allow interactive users to communicate on-line. The MAIL utility is used to send messages to one or more users on a system (or to users on another system via DECnet) in the same way a person would mail a letter. The PHONE utility allows users to communicate interactively in the same way a person would use a telephone.

8.1 USING THE MAIL UTILITY

All mail sent to a user is stored in a file, MAIL.MAI, in the user's top-level directory. This file is accessed by the MAIL utility.

To use the MAIL utility, enter the DCL command MAIL. The mail utility will be invoked, and the MAIL prompt will be output. If the HELP command is entered, all available MAIL commands are listed. Help can be obtained on any of the commands listed by using the HELP facility in the same manner as the DCL HELP facility.

Most MAIL commands ask for the name of the user you are sending the mail to, and the subject of the message. The user name can be preceded by a node name if the user is working on another system.

Examples of user names:

```
Smith
NODEA::Jones
GREAT::Howeser
```

Several MAIL commands will not work unless you are reading or have just read a piece of mail. For example, the FORWARD command forwards the mail just read to the specified user. The discussion of the command output by HELP should be read carefully to notice which commands are in this category.

Table 8-1 lists the major MAIL commands and their functions.

Table 8-1 MAIL commands

Function	Command
Send mail to another user	SEND [file_name]
List all available messages	DIRECTORY
Display a message on the terminal	READ [#]
Copy the current message to the printer	PRINT
Copy the current message to a file	FILE file_name
Send a copy of the current message to another user	FORWARD
Reply to the current message	REPLY
Remove the current message from the mail file	DELETE

When a message is sent to a user, the user is notified by the MAIL utility. A message will appear on the screen, 'new mail from user_name'. The user_name in the message is the name of the user who sent the mail.

If:
- o The user does not read the mail
- o The user is not logged in when the mail is sent

then the MAIL utility keeps track of the number of messages sent. When the user logs in again, the MAIL utility sends a message to his/her terminal indicating the number of messages that have not been read.

To list the available messages, the DIRECTORY command should be used. The READ command accepts a number as a parameter so a specific message can be read. When the user enters MAIL and specifies the READ command without a number, MAIL displays the latest messages received.

```
$MAIL

 MAIL> SEND
 to: Joe Smith
 subj: Sending example of the latest version of GRADES
Enter your message below. Press CTRL/Z when complete, CTRL/C to quit:

 Hi...I am sending you a copy of the latest version of
the GRADES program in the next message for your interest.

^Z

 MAIL> SEND GRADES.FOR
 to: Joe Smith
 subj: Here it is!

 MAIL> EXIT
$
```

To send a file or message to more than one user, list the user names (separated by commas) after the to: prompt, or specify a distribution list. Distribution lists are lists of user names (separated by commas or on separate lines). These lists are stored in files of file_type .DIS. Create a distribution list by using an editor. Use the @ command to specify the file. For example,

```
 Contents of NAMES.DIS:

                   SMITH, JONES, BARKER


$MAIL

 MAIL> SEND Meeting.dat
 to: @NAMES
 subj: Meeting tomorrow

 MAIL> EXIT
$
```

8.2 USING THE PHONE UTILITY

To PHONE another user, enter the PHONE utility by typing the DCL command, PHONE. The information on the terminal screen will be replaced by a screen formatted for the use of PHONE. The PHONE format includes:

- o A command line - beginning with a % prompt.
- o A section of the screen for the caller's use.
- o The lower section of the screen for the callee's use.

The HELP utility in PHONE will list all PHONE commands if HELP is entered on the command line (after the % prompt). Help can be obtained on any PHONE command by entering HELP command.

Users can phone other users, put calls on hold, send short messages using the MAIL utility while in PHONE, send files to other users, and refuse to accept calls. Commands are listed in Table 8-2.

DIAL is the default command. To phone another user, enter DIAL username on the command line (or simply enter the username). Users on other nodes can be dialed via DECnet by specifying the node (node::username).

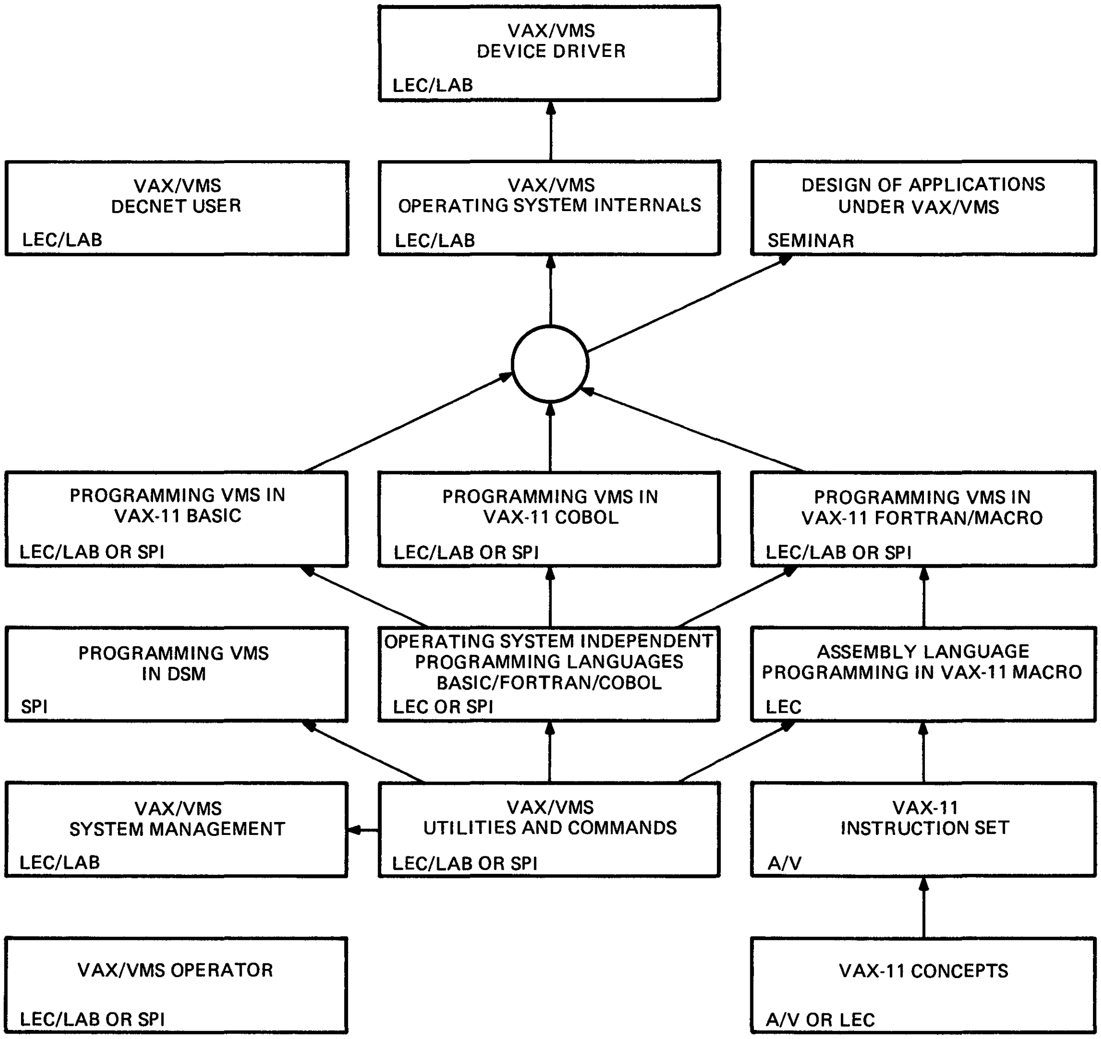
PHONE rings the other users terminal. If the other user enters the DCL command, PHONE, following by the PHONE command, ANSWER, communication can begin. Users enter text which will appear in the top half of their own terminal screen and the bottom half of the other users screen. Several lines of text can be entered. As the user enters text, the text appears on the other user's terminal.

All text entered after the call has begun is assumed to be part of the message. Commands must be entered on the command line only. To get to the command line while entering a message, the switchhook character should be entered. The default switchhook character is the percent sign (%). One command may be entered; then the user is returned to the message area. This is useful for entering commands such as HOLD or REJECT (see Table 8-2).

Table 8-2 PHONE Commands

Function	Command
Place a call (Default)	DIAL username
Answer a call while in PHONE	ANSWER
Display a list of available users (including users on other nodes)	DIRECTORY [node::]
Send the contents of a file to all users involved in the conversation	FACSIMILE file_name
Place a caller on hold	HOLD
Reject a call from a caller	REJECT
Take a caller off hold	UNHOLD
Send a short (one line) message to a user who is unavailable for a PHONE conversation	MAIL
Hangup your own phone	HANGUP (or CTRL-Z)
Obtain help on PHONE commands	HELP

VAX/VMS CUSTOMER CURRICULUM


VAX/VMS DEVICE DRIVER
LEC/LAB
VAX/VMS DECNET USER
LEC/LAB
VAX/VMS OPERATING SYSTEM INTERNALS
LEC/LAB
DESIGN OF APPLICATIONS UNDER VAX/VMS
SEMINAR
PROGRAMMING VMS IN VAX-11 BASIC
LEC/LAB OR SPI
PROGRAMMING VMS IN VAX-11 COBOL
LEC/LAB OR SPI
PROGRAMMING VMS IN VAX-11 FORTRAN/MACRO
LEC/LAB OR SPI
PROGRAMMING VMS IN DSM
SPI
OPERATING SYSTEM INDEPENDENT PROGRAMMING LANGUAGES BASIC/FORTRAN/COBOL
LEC OR SPI
ASSEMBLY LANGUAGE PROGRAMMING IN VAX-11 MACRO
LEC
VAX/VMS SYSTEM MANAGEMENT
LEC/LAB
VAX/VMS UTILITIES AND COMMANDS
LEC/LAB OR SPI
VAX-11 INSTRUCTION SET
A/V
VAX/VMS OPERATOR
LEC/LAB OR SPI
VAX-11 CONCEPTS
A/V OR LEC

KEY

LEC	=	LECTURE
LEC/LAB	=	LECTURE AND LAB
SPI	=	SELF-PACED INSTRUCTION
A/V	=	AUDIOVISUAL INSTRUCTION

TK-9040

For more information concerning VAX/VMS Education, contact your Educational Services Marketing Representative, Digital Sales Representative or your nearest Digital Training Center.

Australia:
Digital Equipment Australia Pty Ltd.
Educational Services Department
Chatswood Plaza Building
P.O. Box 384
Chatswood, New South Wales, 2067
Telephone: (02) 412 5252

Canada:
Digital Equipment of Canada Ltd.
Educational Services Department
100 Herzberg Road
P.O. Box 13000
Kanata, Ontario K2K 2A6
Telephone: (613) 592 5111

Digital Equipment of Canada Ltd.
Educational Services Department
165 Attwell Road
Rexdale, Ontario M9W 5Y5
Telephone: (416) 674 2580

Digital Equipment of Canada Ltd.
Educational Services Department
10711 Cambie Road, Suite 130
Richmond, British Columbia
V6X 3C9
Telephone: (604) 278 3466

Digital Equipment of Canada Ltd.
Educational Services Department
394 Isabey Street
St.-Laurent, Quebec
H4T 1V3
Telephone: (514) 342 5321

Europe:

Belgium
Digital Equipment N.V.-S.A.
Educational Services Department
Boulevard Brand Whitlock 87
B-1040 Brussels
Telephone: [32]-(2)-733-9650

England
Digital Equipment Co. Ltd.
Educational Services Department
Fountain House, The Butts Center
Reading RG1 7QN
Telephone: [44]-(734)-583555

Digital Equipment Co. Ltd.
Education Services Department
Arndale House
Chester Road, Stretford
Manchester M32 9BH
Telephone: [44]-(61)-865-0785

Finland
Digital Equipment Corporation OY
Educational Services Department
P.O. Box 16
SF-02201 Espoo 20
Telephone: [358]-(0)-423511

France
Digital Equipment France
Service Education
2 rue Gaston Cremieux
Evry les Epinettes
BP 136
F-9100 Evry Cedex
Telephone: [33]-(6)-077-8292

Ireland
Digital Equipment Ireland Ltd.
Educational Services Department
Park House
North Circular Road
Dublin 7
Telephone: [353]-(1)-308-433

Italy
Digital Equipment Corporation S.p.A.
Educational Services Department
Viale Fulvio Testi 117
I-20092 Cinisello Balsamo
Milam
Telephone: [39]-(2)-61797

Netherlands
Digital Equipment BV
Educational Services
Ratelaar 38
3434 EQ Nieuwegein
Telephone: [31]-(3402)-45654

Spain
Digital Equipment Corporation S.A.
Educational Services Department
Agustin de Foxa, 27
Madrid 16
Telephone: [34]-(1)-733-1900

Sweden
Digital Equipment AB
Educational Services Department
Box 1250
S-171 24 Solna
Telephone: [46]-(8)-7300200

Switzerland
Digital Equipment Corporation, SA
Educational Services Department
Schaffhauserstrasse 144
CH-8302 Kloten/ZH
Telephone: [41]-(1)-8169111

West Germany
Digital Equipment GmbH
Educational Services Department
Wallensteinplatz 2
D-8000 Munich 40
Telephone: [49]-(89)-35030

Japan:
Digital Equipment Corporation Int.l
Educational Services Department
Sunshine 60, P.O. Box 1135 36th floor
1-1 Higashi Ikebukuro 3-Chome
Toshima-Ku, Tokyo 170, Japan
Telephone: (03) 989 7180

Digital Equipment Corporation Int'l
Educational Services Department
Koei Building Shinkan 4F
3-7 Nishitenma 6-Chome
Kitaku, Osaka 530, Japan
Telephone: (06) 364 0401

Mexico:
Digital Equipment de Mexico,
Educational Services Department
Nueva York 115, Col. Napoles
03810 Mexico, D.F.
Telephone: (905) 687 6681

United States:

Boston
Digital Equipment Corporation
Educational Services Department
12 Crosby Drive
Bedford, Massachusetts 01730
Telephone: (617) 276 4111

Chicago
Digital Equipment Corporation
Educational Services Department
5600 Apollo Drive
Rolling Meadows, Illinois 60008
Telephone: (312) 640 5520

Dallas
Digital Equipment Corporation
Educational Services Department
12100 Ford Road
Suite 110
Dallas, Texas 75234
Telephone: (214) 620 2051

Los Angeles
Digital Equipment Corporation
Educational Services Department
4311 Wilshire Boulevard
Suite 400
Los Angeles, California 90010
Telephone: (213) 937 3870

New York
Digital Equipment Corporation
Educational Services Department
One Penn Plaza
New York, New York 10001
Telephone: (212) 971 3545

San Francisco
Digital Equipment Corporation
Educational Services Department
2525 Augustine Drive
Santa Clara, California 95051
Telephone: (408) 727 0200

Washington
Digital Equipment Corporation
Educational Services Department
Lanham 30 Office Building
5900 Princess Garden Parkway
Lanham, Maryland 20801
Telephone: (301) 459 7900

digital
25
1957-1982

EDUCATIONAL SERVICES