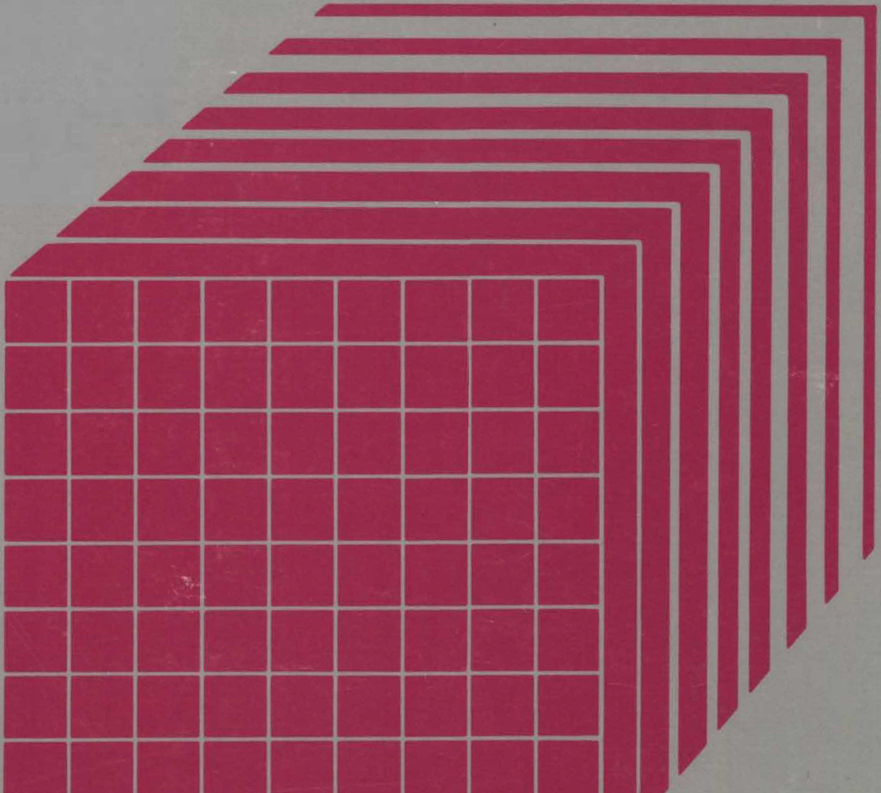
IBM

Virtual Machine/
System Product

Application Development Guide

Release 5

SC24-5247-2



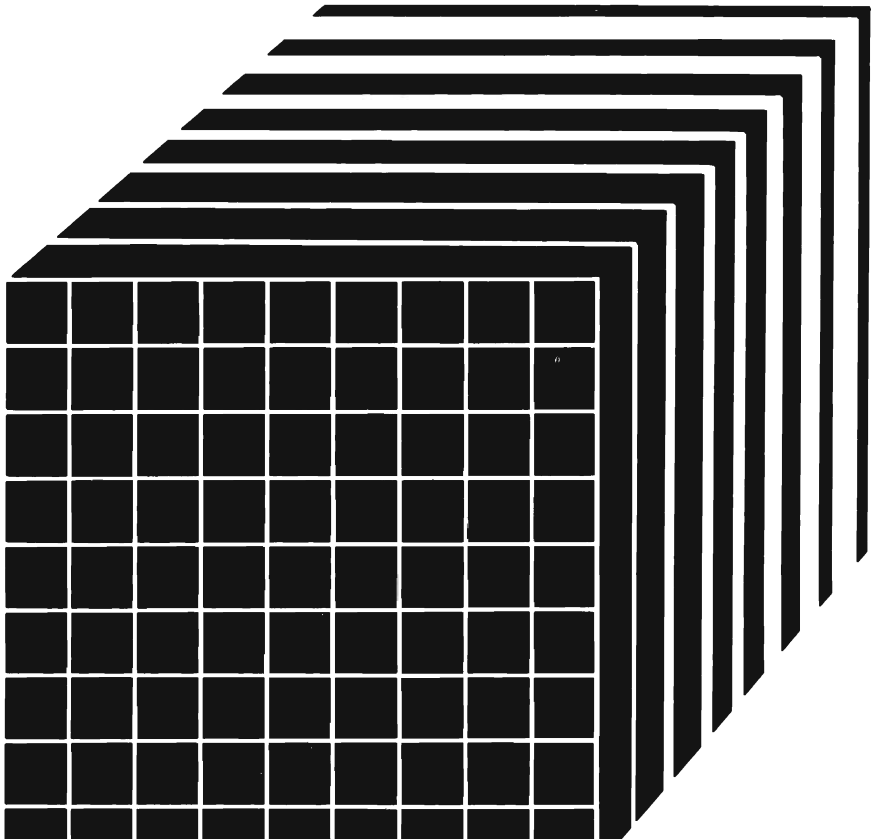
IBM

Virtual Machine/ System Product

Application Development Guide

Release 5

SC24-5247-2



Third Edition (December 1986)

This edition, SC24-5247-2, applies to Release 5 of the IBM Virtual Machine/System Product, (VM/SP), Program Number 5664-167, and to all subsequent releases and modifications until otherwise indicated in new editions or Technical Newsletters. Changes are made periodically to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370, 30xx, and 4300 Processors Bibliography,* GC20-0001, for the editions that are applicable and current.

Summary of Changes

For a list of changes, see page 287.

Ordering Publications

Requests for publications should be made to your IBM representative or the IBM branch office serving your locality. Publications are *not* stocked at the address given below.

A form for reader's comments is provided in this publication. If the form has been removed, comments may be addressed to IBM Corporation, Information Development, Dept. G60, P.O. Box 6, Endicott, NY, U.S.A. 13760. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Preface

This manual is an introduction to developing and running COBOL and FORTRAN application programs under VM/SP.

This manual is designed for experienced COBOL or FORTRAN programmers who are unfamiliar with VM/SP.

Before reading this book you may want to read *VM/SP CMS Primer*, SC24-5236.

After studying the material in this manual, you'll be able to:

- Log on to VM/SP.
- Use the editor to enter and modify program source statements.
- Save the statements in a file.
- Use a language compiler.
- View, print, and save the output from the compiler as LISTING and TEXT files.
- Run and test a program using the test tools of VM/SP and the language processor.

This book tells how to:

- Use ISPF or DMS/CMS to design and manage dialogs and dialog screens.
- Use the data base management system facilities of SQL/DS in an application program or in an EXEC.
- Prototype applications using the System Product Interpreter.
- Use the Vector Facility support furnished by VM/SP HPO.
- Debug your application program.

This is not a reference manual. After working through this book, you are expected to consult appropriate manuals for more information on specific subjects. For a list of books, see "Bibliography" on page 289.

In Chapters 1 through 11, you'll learn:

Chapter 1	How to log on and off the system, and enter VM/SP commands.
Chapter 2	How to develop programs using CMS.
Chapter 3	How to create source programs and other files.
Chapter 4	How to compile, link-edit, and execute your programs.
Chapter 5	How to make use of the various CMS libraries.
Chapter 6	How to invoke dialog management from within the application.
Chapter 7	How to invoke data base facilities from within the application.
Chapter 8	How to use the System Product Interpreter (the Restructured Extended Executor Language).
Chapter 9	How to use special features in CMS that can make certain programming tasks easier.
Chapter 10	How to test and debug your programs in CMS.
Chapter 11	How to use the VM/SP HPO Vector Facility support with FORTRAN programs.

See "Introducing VM/SP" on page 1 for a general overview of VM/SP.

Contents

Figures

Introducing VM/SP

This section is an overview of the VM/SP system from an application programmer's point of view. When you finish, you should have an understanding of what VM/SP is, how it's structured, and its major features and facilities.

Virtual Machine/System Product (VM/SP), or just **VM**, is an operating system that lets you and many other users each appear to have a complete computing system environment at your disposal. This means that the computer system itself (the hardware — CPU, disk drives, tape drives, printers, etc.) can support a large number of users who all need the machine at the same time.

VM allocates machine resources (hardware, storage and processing time) so that you **appear** to have control over an entire self-contained, private computing system, as in Figure 1 on page 2.

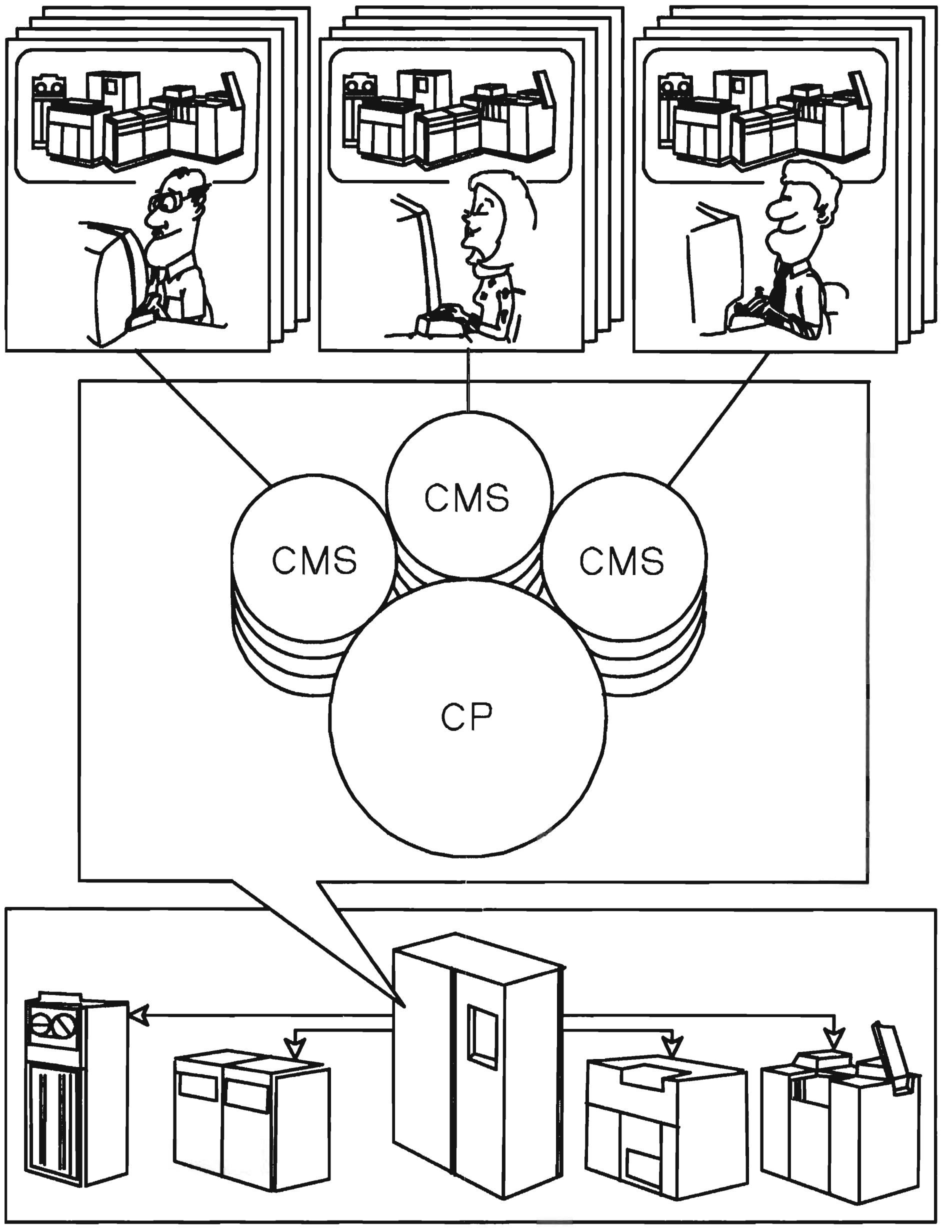


Figure 1. CP and CMS Allow VM/SP Users to Share One System

Because you don't really have direct control over the real machine, your configuration is known as a **virtual machine**. Each virtual machine operates in the real computer under control of a part of the VM environment known as the **Control Program (CP)**. The Control Program is the resource manager for the real computer. It ensures that each virtual machine is allocated the resources it needs to perform its own jobs.

Among the operating systems that can run in a virtual machine are DOS/VS, MVS, and CMS. This means, for example, that an entire MVS system can run within the VM environment. It runs exactly as it would on a real machine, complete with support for multiple TSO users, batch jobs, etc. VM can support multiple virtual MVS machines, virtual DOS/VS or VSE/AF machines, and a large number of users each running VM's own interactive operating system, **Conversational Monitor System (CMS)**.

CMS is a virtual machine operating system that provides functions for you to use at the terminal. It's specifically designed to run in the VM environment and depends on CP for its execution. Thus, CMS can't operate independently on a real machine as can other operating systems.

CMS provides an individually tailored computing environment. The CMS environment is complete with commands, a file system, and terminal communication facilities. CMS is an interactive environment. You invoke CMS functions through commands entered at your terminal. Because it's an operating system, you can also run programs written in standard programming languages such as Assembler, FORTRAN and COBOL. CMS is designed to make the whole programming process easier: program design, development, testing, and implementation.

You enter the CMS environment from the CP environment automatically (if your system has automatic IPL), or by issuing the **IPL** command (see Loading CMS on page 15.) The **IPL** command loads CMS into your virtual machine. If you're planning to use CMS for your entire terminal session, you shouldn't have to IPL again unless a program failure forces you into the CP environment.

You can tell which environment you're in (with the exception of the input mode of the edit environment) by pressing the RETURN or ENTER key on a null line (that is, a line that has nothing keyed in on it). If the system responds by displaying the prompt "CMS" it means that you're in the CMS environment. If the system responds by displaying the prompt "CP" it means that you're in the CP environment. If the system responds by displaying the prompt "GCS" it means that you're in the GCS environment.

When your virtual machine is in the CMS environment, you can issue any CMS command or CP command valid for your user privilege class.

You can execute many language processors under CMS, including:

The Assembler
IBM BASIC
VS BASIC
VS APL
OS FORTRAN
VS FORTRAN
VS FORTRAN Version 2
OS/VS COBOL
OS PL/I Optimizing and Checkout Compilers

The **HELP** command displays information on how to use CP commands and CMS commands, subcommands, EXECs, and explanations of VM messages.

More About CMS

Like all operating systems, CMS has a well-defined system structure. The three parts of CMS (Terminal System, File System, and CMS System Services) are shown in Figure 2.

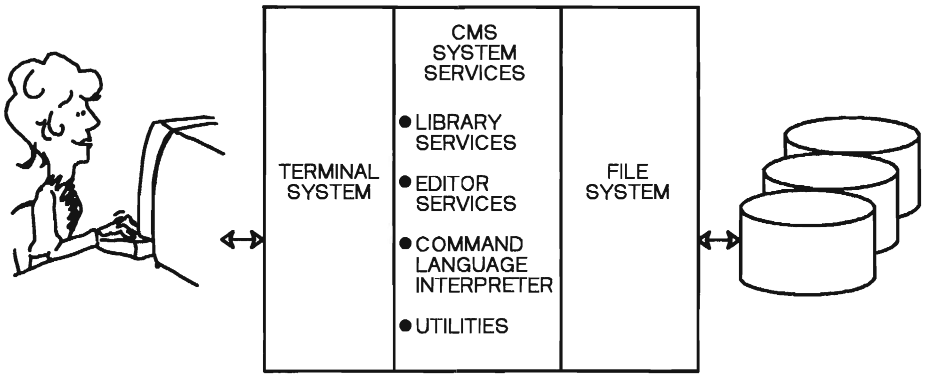


Figure 2. The CMS System Structure. This consists of the Terminal System, CMS System Services, and File System.

Terminal System
This is a portion of the CMS system that supports your terminal. It reads commands entered at the keyboard and displays system responses to those commands.

System Services
This is a portion of CMS that constitutes the basic user interface. It consists of a number of distinct **facilities**, such as:

- Library Services
- Utility Commands
- The System Product Editor
- The System Product Interpreter.

File System
This provides basic input and output services, such as **read** and **write** operations. These I/O functions are used by the system services and also by user programs running in the CMS virtual machine.

Each of these has a number of commands that invoke particular CMS features. Each makes use of the terminal system and file system portions of CMS. Simply enter the various commands at the keyboard and you'll see the results displayed on the terminal.

We'll discuss the File System and System Services in a little more detail.

System Services

Most operating systems provide **library** facilities. These help you develop programs and maintain an orderly environment for managing your files. A library is a special type of CMS file that groups files (known as **members**) of a similar nature and function. To manipulate libraries and their members, you can use the library facilities, which are operating system functions. For example, you can use the following commands:

MACLIB	creates or changes a library of assembly language macros or high-level language COPY code.
TXTLIB	creates or changes a library of program object code.
GLOBAL	defines certain kinds of **libraries** used by compilers or by CMS itself when loading a program before running it.

Utility Commands

Besides the commands you use for handling files and manipulating libraries, there are a number of commands that let you reconfigure your virtual machine, communicate with other users, and control program development and implementation. For example, you can use the following commands to reconfigure your minidisks:

ACCESS	changes the order in which CMS searches minidisks for files or gains access to a minidisk not yet available to CMS.
RELEASE	makes a minidisk unavailable without affecting its contents.

You can use the following commands to communicate with other users:

SENDFILE	lets you send one or more files to another user.
RECEIVE	lets you receive a file sent to you by another user.
TELL	lets you send a one-line message to another user.
NOTE	lets you send a longer message to another user.

TELL and **NOTE** make use of a special file containing the userids of the virtual machines of other VM users. It also contains other information, such as their first and last names and even nicknames. You can create this file by using the **NAMES** command.

Other utility commands let you control program development and implementation. For example, after you have written your program using the System Product Editor, you'll need to compile it. Then, you'll want to run the program and maybe debug it. You can use the following utility commands to accomplish these tasks:

COBOL	compiles a COBOL program, using the COBOL/VS compiler.
FORTVS2	compiles a FORTRAN program, using the VS FORTRAN Version 2 compiler.
LOAD	link-edits a program and loads it into storage, ready for execution.
START	initiates execution of a program loaded into storage.
DEBUG	lets you debug a running program by displaying and altering part of storage.
TESTCOB	lets you debug a COBOL program interactively.
TESTFORT	lets you debug a FORTRAN program interactively.

The System Product Editor

With the System Product Editor (or just "the editor") you can enter and create files that will reside on a CMS minidisk. Use the **XEDIT** command to call the editor and create source programs, data files, documentation files, or special files (for example, source program update files, message files, and display panel definitions). The editor lets you insert, delete, copy, and relocate lines of code or data, alter character strings on a single line or throughout a file or portion of a file, search for character strings, sort lines within a data file, and the like.

You can also:

- Edit more than one file at a time.
- Display more than one image or file on the terminal screen.
- Restrict the display to selected lines (for example, only those with a certain character string).
- Display key portions of the screen in various colors (on terminals supporting multiple-color displays).

You can display more than one image at a time to compare two files (displayed side-by-side) or debug a source program. In debugging, you can set up a split screen with the source program displayed on the top of the screen and a listing of compilation errors on the bottom. You can then debug your program by noting the list of errors on the bottom and correcting them in the source displayed on top. Thus, you may not need a hardcopy listing.

You can also use the CMS editor to develop edit macros, groups of edit subcommands executed sequentially in a known sequence but invoked by one command. For example, you can write an edit macro to insert an EXIT statement in a COBOL program whenever invoked or at key places in a file, such as following the character string -EXIT every time it occurs. For details on how to use XEDIT and its subcommands, see "Step 1: Create a File" on page 19. For further information, see *VM/SP System Product Editor Command and Macro Reference.*

The System Product Interpreter

The System Product Interpreter is another powerful CMS tool that lets you execute sequences of CMS commands in the same way the CMS Editor macro facility lets you execute a sequence of **XEDIT** subcommands. Using the interpreter, you can, for example, create a single command (called, in this case, an EXEC) that compiles, link-edits, and executes an existing COBOL or FORTRAN program. You can further enhance such an EXEC to first invoke the CMS editor. And, if compilation errors occur, you can reenter the editor to correct them, only going on to link edit and run the program if no compilation errors occur.

You can use the Restructured Extended Executor language, which is processed by the interpreter to write EXECs to prototype applications. With the interpreter, you can develop and test algorithms before coding them in a high-level language. Since standard programming structures such as **If-Then-Else, Do-While, SELECT-WHEN,** and **Do-Until** are included in the executor language, you can easily translate one of its routines into a high-level language. "Chapter 8: EXECs" on page 191 gives more information on how to write an EXEC.

The File System

One unique feature of the VM file system is the **minidisk**. A diagram of this is shown in Figure 3 on page 8. Through the use of minidisks, Direct Access Storage Device (DASD) space is allocated to you without having to dedicate an entire DASD pack to you (unless all its space is needed). At the same time, CMS maintains a secure and integral file system for the virtual machine. When a virtual machine is defined to VM, disk space is allocated in contiguous cylinders or blocks (depending on device type). Thus, there can be on or more cylinder (or blocks) of DASD space allocated for the exclusive use of your virtual machine.

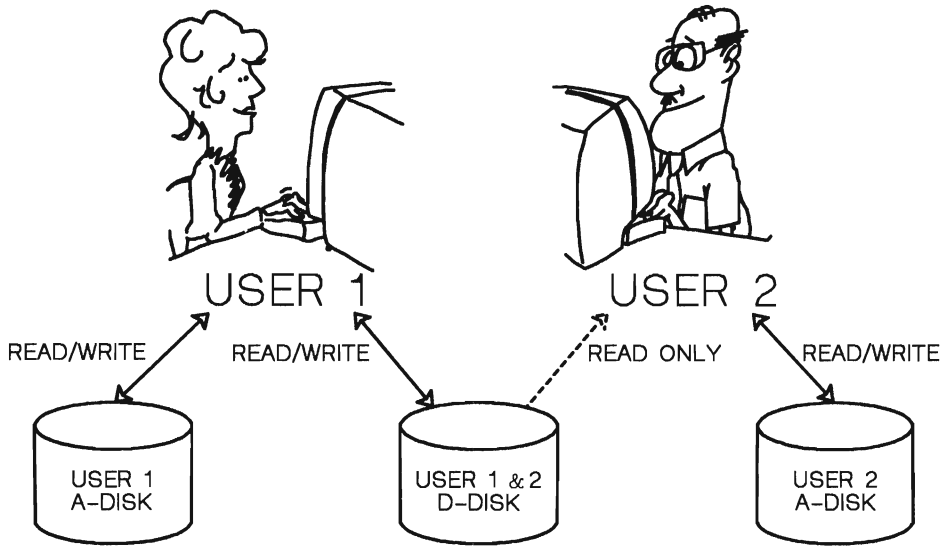


Figure 3. Users Can Share Disks in the "Read Only" Mode and Own Disks in the "Read/Write" Mode

CMS support is totally responsible for file management, including blocking and deblocking. You manipulate files by name. Individual file space is not preallocated. It is obtained and deleted dynamically from your block of allocated space. You can query the disk at any point to determine the amount of free space.

Once disk space has been allocated, it's formatted to comply with the blocking structure of CMS and may then be used to contain CMS files that you have created. Actually, your virtual machine may have several minidisks defined to it. You can access up to 26 minidisks at a time.

The number 26 suggests letters of the alphabet, and this is in fact the way in which CMS identifies your minidisks. This letter, known as the **filemode**, becomes part of the identifier for each file on a given minidisk.

Each file is uniquely identified by this **file identifier**, which consists of three parts: **filename**, **filetype**, and **filemode**. When you create a file, the **filename**, **filetype**, and **filemode** are assigned by you. The filename and filetype can be up to eight characters long.

Except for some standard conventions (such as the CMS system disk), you're in complete control of the **filename** and **filetype** of a minidisk file and how CMS should access them.

COBOL and FORTRAN compilers **require** corresponding filetypes for their respective programs. Use COBOL or FORTRAN as the filetypes of programs you write in these languages.

CMS has many commands that address files through the file identifier or **fileid**. For example, you can **COPY** a file from one minidisk to another (or to the same minidisk, using a new fileid), or **RENAME** a file, changing the filename, the filetype, the filemode or all three. You can also create a new file or modify an old one by using the **XEDIT** command, which invokes the standard CMS editor. You can **ERASE** a file you don't need. You can **PRINT** a file. If you want to see all or part of a file without invoking the editor, you can **TYPE** the file, which displays the contents on your terminal. To see the files you have on one or more of your minidisks, you can use the **FILELIST** command.

Each command has a syntax corresponding to its function. You can find all CMS commands listed in alphabetic order in *VM/SP CMS Command Reference*, along with their rules and associated messages.

Summary

The basic three-part structure of the CMS operating system simplifies your programming process by providing you with:

- A terminal interface for line-edit and full-screen operation.
- A file access method to ensure the security and integrity of your files.
- A set of system services with utility commands, library services, an editor, and the Command Interpreter.

We have examined the VM/SP system from an application programmer's point of view. VM manages the resources of a real machine so that you have the functional equivalent of a computing system complete with terminal, unit record, and DASD devices as needed. (VM/SP also supports tape devices and other special devices that may be used occasionally.)

We've also looked briefly at VM's unique operating system, CMS. This system provides terminal support, a file system, and a conversational command interface with a wide variety of functions. They range from program design, coding, testing, debugging, implementation, and documentation to such special features as screen dialog management and data base access.

Finally, we have seen that VM lets you share data with other users as well as communicate with them.

Chapter 1: Getting Started with VM/SP

This chapter tells how to log on to VM/SP, begin your terminal session, and log off. It discusses your keyboard and shows you how to recognize the terminal status your system may be in.

Your Keyboard

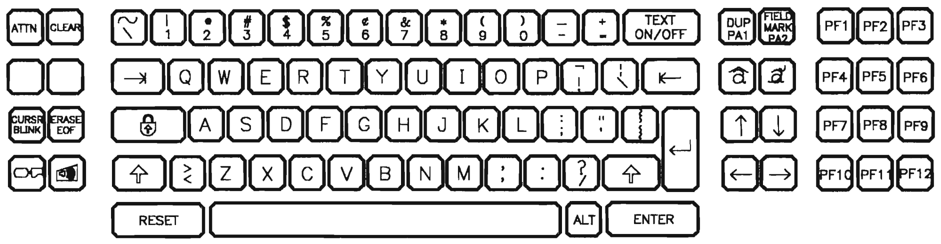


Figure 4. An Example of a Keyboard Layout

The keyboard you are using is composed of the following:

Character Keys, which include:

- Alphabetic (A through Z)
- Numeric (0 through 9)
- Punctuation characters such as **!**, **,**, **:**, **;**, and **?**.
- Special characters such as **@**, **$**, **%**, and *****.
- Text characters — if your keyboard is operating in the text mode, press and hold the CODE key, then press a character key to obtain the text character that is engraved on its front face. For more information, see *VM/SP Terminal Reference.*

Control Keys, which include:

- Program Function (PF) Keys.

  You can define these keys to have either command or data-input capability by using the **SET PFnn** command. For information on this command, see *VM/SP CP Command Reference.*

- Cursor Control Keys.
- Screen Management Keys, which include:

CLEAR	clears the entire screen--output area, input area, and status area.
ERASE INPUT	erases the user-input area.
ERASE EOF	erases from the cursor to the end of the line.
PA1	posts an attention interruption pending to the CP command environment. When working in full-screen CMS (with SET FULLSCREEN ON), the PA1 key no longer serves as an ATTENTION key, it performs a windowing function.
PA2 or CNCL	clears the output display area.
INS MODE	Press this key to enter the **insert mode**.
DEL	deletes the character indicated at the cursor and shifts the data line one space to the left.

Beginning Your Terminal Session

To establish contact with VM/SP, switch your terminal on. VM/SP should respond with a screen displaying a message:

```
VIRTUAL MACHINE/SYSTEM PRODUCT
```

This lets you know that VM/SP is running and that you can use it. If you don't receive the **VM/SP ONLINE** message, see *VM/SP Terminal Reference* for further instructions.

Before you can use VM/SP, you must identify yourself by giving your userid and password:

userid	a symbol (eight characters or less) that identifies your virtual machine to VM/SP and lets you gain access to the system.
password	a symbol (eight characters or less) that functions as a protective device. No one can use your virtual machine unless they know your password.

To get a userid and password, see your supervisor.

Note: Different installations have somewhat different ways of establishing contact with VM/SP. The procedure at your installation may vary from the procedure described here. Check with your supervisor or system administrator.

How to Log On to VM/SP

Note: If your terminal is not a 3270-type, use the logon procedure described in "Logon Exception."

Logging On from a 3270-type Terminal

When you have the VM/SP logo screen displayed, you are ready to initiate logon processing.

If your terminal is a 3270-type, you may log on directly from the logo screen. Below the actual VM/SP logo are two lines instructing you to fill in your userid and password. Following these instructions are three input lines labeled USERID, PASSWORD, and COMMAND. The cursor is placed at the input line for USERID.

You may now type your userid and password in the USERID and PASSWORD input areas and press ENTER.

Note: In the rest of this manual, we'll use **enter** to mean that you should type in the line or lines indicated and then press the ENTER key.

If all of the information is entered correctly, the logo is cleared from the screen, no further prompts will appear, and you will be logged onto the system. If an invalid userid or password is entered, the logo is cleared from the screen, and the following message and prompt will appear:

```
DMKLOG050E LOGON UNSUCCESSFUL - INCORRECT PASSWORD
or
DMKLOG053E userid NOT IN CP DIRECTORY

Enter one of the following commands on the COMMAND line below:

   LOGON userid              (Example: LOGON VMUSER1)
   DIAL userid               (Example: DIAL VMUSER2)
   MSG userid message        (Example: MSG VMUSER2 GOOD MORNING)
   LOGOFF
```

If you enter only your PASSWORD in the input area, the following error message will be issued, followed by the LOGON prompts:

```
DMKCFM288E LOGON FROM THE INITIAL SCREEN WAS UNSUCCESSFUL
```

If your USERID, as entered, contains one or more blanks (V MUSER1), the following error message will be issued, followed by the LOGON prompts:

```
DMKLOG053E V NOT IN CP DIRECTORY
```

You may also enter your userid in the USERID input area without your password or enter the **LOGON** command, followed by your userid, in the COMMAND input area. The following prompt will appear:

```
ENTER PASSWORD (IT WILL NOT APPEAR WHEN TYPED):
```

If your installation permits, you may enter the LOGON command followed by your USERID and PASSWORD in the COMMAND input area.

```
COMMAND  ===>Logon VMUSER1 password
```

If you have entered the information correctly, the logo is cleared from the screen and you will be logged onto the system.

Logon Exceptions

If your terminal is not a 3270-type, and you have the VM/SP logo screen displayed, you can now press the ENTER key on your terminal to clear the screen and initiate logon processing.

Now enter the **LOGON** command. If, for example, your userid is **SMITH**, then type:

```
logon smith
```

and press the **ENTER** key. The short form of **LOGON** is **L**, so you can just enter l **smith**.

Note: You can enter commands using any combination of upper-case and lower-case characters; VM/SP translates your input to upper-case. Examples in this publication show all user-entered input lines in lower-case characters and all system responses in upper-case characters.

If VM/SP accepts your userid, it responds by asking for your password:

```
ENTER PASSWORD
```

Carefully type your password, and then press the **ENTER** key. *For reasons of security you won't see your password as you type it.* If you receive the message:

```
PASSWORD INCORRECT
```

you'll have to start over, beginning with the **LOGON** command.

After you key in the proper identification, press the **RETURN** or **ENTER** key.

If the logon procedure has been successful, VM establishes a virtual machine in the system for your use. While doing so, it displays progress messages on the screen.

Loading CMS

If your virtual machine has been set up to automatically **Initial Program Load (IPL)** CMS for you, you'll get a message that looks something like this:

```
VM/SP CMS - 05/16/84  11:45
```

The terminal status displayed is VM READ. When you press ENTER, the remainder of your virtual machine facilities are set up and a message like this is displayed:

```
Ready; T=0.01/0.01 11:15:30
```

This is called the **ready message**. If AUTOCR is specified in your directory entry, you do not have to press ENTER.

If your terminal status shows:

```
CP READ
```

after you enter your password, CMS has not been loaded. This can happen after a CMS system failure, or because your userid has not been properly set up by the system administrator. You'll have to IPL CMS yourself. Enter:

```
ipl cms
```

When VM READ appears as the terminal status, press ENTER again. VM responds with a ready message, as shown above, indicating that CMS is ready to receive commands.

Note: If you get the message:

```
DMSACC112S A(191) DEVICE ERROR
```

you must **format** your virtual disk for use with CMS files. To help you do this, see your supervisor or system administrator.

How to Log Off

To end your terminal session, use the **LOGOFF** command. Enter:

```
logoff
```

The short form for **LOGOFF** is **LOG**.

Terminal Status Notices

The following is explanation of all the **terminal status** messages that may appear at the lower right corner of your screen.

CP READD

The Control Program issued a read request to your terminal and is waiting for a reply. After you log on, this is the first status notice you see. It also occurs, for example, after a message that requires a response. If you type in your reply and press **ENTER**, processing will continue. If you see this message when you don't expect it, enter:

```
B
```

If this doesn't work, IPL the system again by typing:

```
ipl cms
```

then press enter, wait for the system response, and then press enter again.

VM READ

The operating system running in your virtual machine issued a read request to your terminal and is waiting for a reply. If you type in your reply and press **ENTER**, processing will continue.

RUNNING

CP or your virtual machine is working on something, or is waiting for you to enter a command. **RUNNING** can also occur if the screen is filled and there are no additional lines to display. Once you've loaded CMS and are using the CMS environment, this status is almost continually in effect.

MORE...

CP or your virtual machine is running, but the output display area is full and there are more lines of output to be displayed. When you see the screen is in this status, you can do one of the following:

- Press the **CLEAR, CANCEL**, or **PA2** keys to clear the screen and see the next screen.
- Press the **ENTER** key to hold the screen in its present status. This changes the status to **HOLDING**.

If you don't do either, after 60 seconds, the screen is cleared and the next screen is displayed.

HOLDING

This status appears when the screen displayed a **MORE...** notice and you pressed the **ENTER** key. CP or your virtual machine is running and the screen is full. This notice can also appear when another user sends you a message. To end the hold, press the **CLEAR** key.

NOT ACCEPTED You typed a command or a line of data and pressed **ENTER**, but the terminal buffer is full and can't accept it. VM/SP locks the keyboard for about three seconds while it displays the **NOT ACCEPTED** status, then reverts to the previous status.

The rejected data stays in the user input area of the screen so you can retry the operation without typing it again. Just press **ENTER** after the **NOT ACCEPTED** notice goes away.

When you are working in full-screen CMS (with SET FULLSCREEN ON), the following status messages will be displayed.

Note: For a complete explanation of the SET FULLSCREEN ON command, refer to the *CMS Command Reference*.

Executing a command: The system is processing your command.

Enter your response in vscreen *'vname':* The system is waiting for you to reply to a request.

Note: In this message vname will be replaced by the name of the virtual screen in which you are to enter your response.

Scroll for more information in vscreen *'vname':* To see the waiting information you must scroll forward a window that is showing the specified vscreen.

Enter a command or press a PF or PA key: The system is waiting to process your next input.

Note: In this message vname will be replaced by the name of a virtual screen. You must scroll forward a window connected to the virtual screen in order to see the waiting information.

Summary

In this chapter, you learned how to log on to VM and begin your terminal session, and how to log off. You became familiar with your keyboard and learned how to recognize the terminal status of your system.

Chapter 2: Developing Programs Using CMS

In this chapter, you'll learn how to create a file, invoke the System Product Editor, enter a COBOL or FORTRAN source program, save the program, compile the program, and execute it.

We'll do this in three steps:

1. Create a file containing a program.
2. Compile the program.
3. Run the program.

If you're not already logged on, do so now. The log on procedure is described in "How to Log On to VM/SP" on page 13.

Step 1: Create a File

The **System Product Editor**, which we'll also refer to simply as . **the editor**, is a full-screen editor that creates or modifies files.

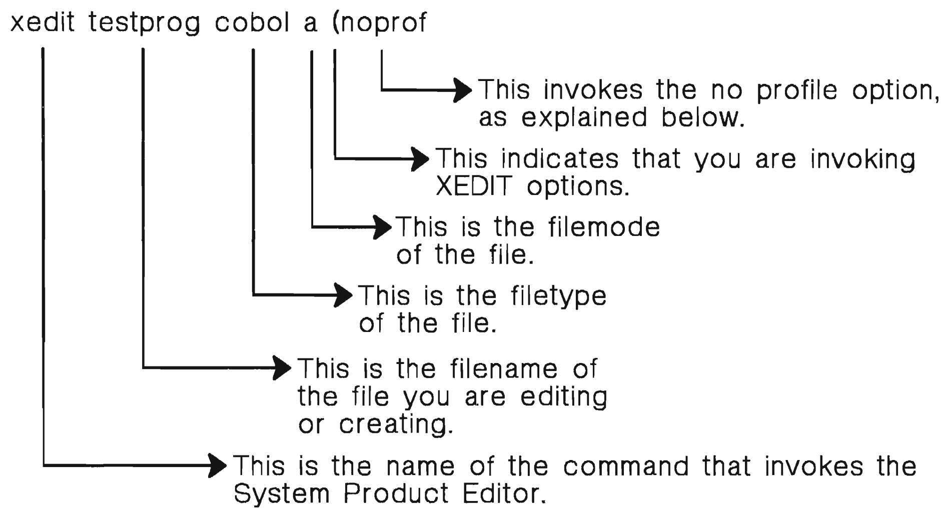
In this chapter you're going to use the System Product Editor to create a file, enter a program, and then file it. Modifying the file and manipulating various options are discussed in "Chapter 3: Using the System Product Editor" on page 39.

If you're programming in FORTRAN, skip to "Creating a FORTRAN File" on page 24.

Creating a COBOL File

You invoke the System Product Editor by entering the **XEDIT** command, followed by the file identifier of the file you want to create or edit. If there are options you want to apply to this session, put them next, after an open parenthesis.

For example, your **XEDIT** command might look like this:



```
xedit testprog cobol a (noprof
```

This invokes the no profile option, as explained below.
This indicates that you are invoking XEDIT options.
This is the filemode of the file.
This is the filetype of the file.
This is the filename of the file you are editing or creating.
This is the name of the command that invokes the System Product Editor.

Note: If you plan to go through this book doing both the COBOL and FORTRAN examples, make sure you give the two files you create different filenames (that is, you must change **testprog** to something else for the second language).

In our example we'll use a file with the **file identifier** TESTPROG COBOL A. The **filename** (TESTPROG), and the **filetype** (COBOL), are parts of the identifier you assign when you create the file. The **filemode** (A) is automatically assigned by CMS.

Issue the **XEDIT** command now. Enter:

```
xedit testprog cobol a (noprof
```

(You can use **X** as an abbreviation for **XEDIT**.)

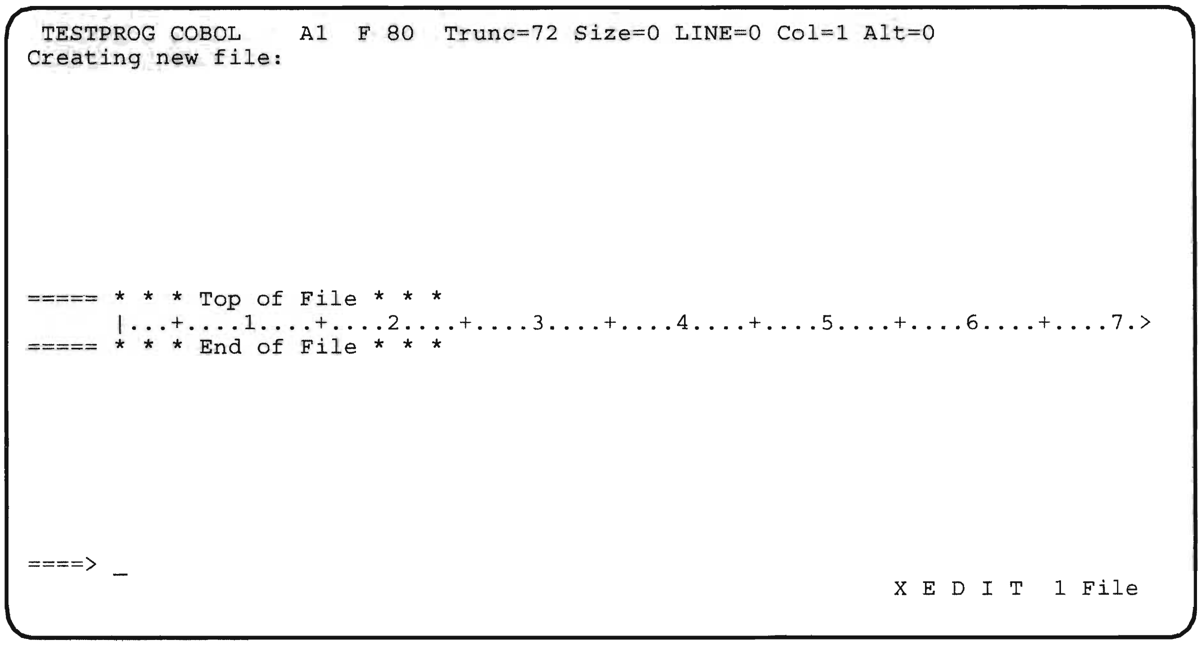
Note: We're using the **noprof** (no profile) option to make sure that your screen looks as shown in this manual, no matter how your installation has customized the programmer environment. Normally you'd enter the command:

```
xedit testprog cobol a
```

or just

```
x testprog cobol a
```

Your display screen should look like this:



```
 TESTPROG COBOL    A1  F 80  Trunc=72 Size=0 LINE=0 Col=1 Alt=0
Creating new file:








===== * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== * * * End of File * * *








====> _
                                                          X E D I T  1 File
```

Here are what the various parts of the screen mean:

TESTPROG	is the filename.
COBOL	is the filetype.
A1	is the filemode.
F	is the type of file format, where: • F = fixed length records • V = variable length records
80	is the record length.
Trunc=72	indicates when truncation begins. If the number is less than the record length, data entered at this column or beyond is ignored.
Size=0	is the number of records in the file.
Line=0	indicates the line number of the current line.
Col=1	indicates the position of the column pointer. The column pointer is the vertical bar (|) in column 1 of the scale.

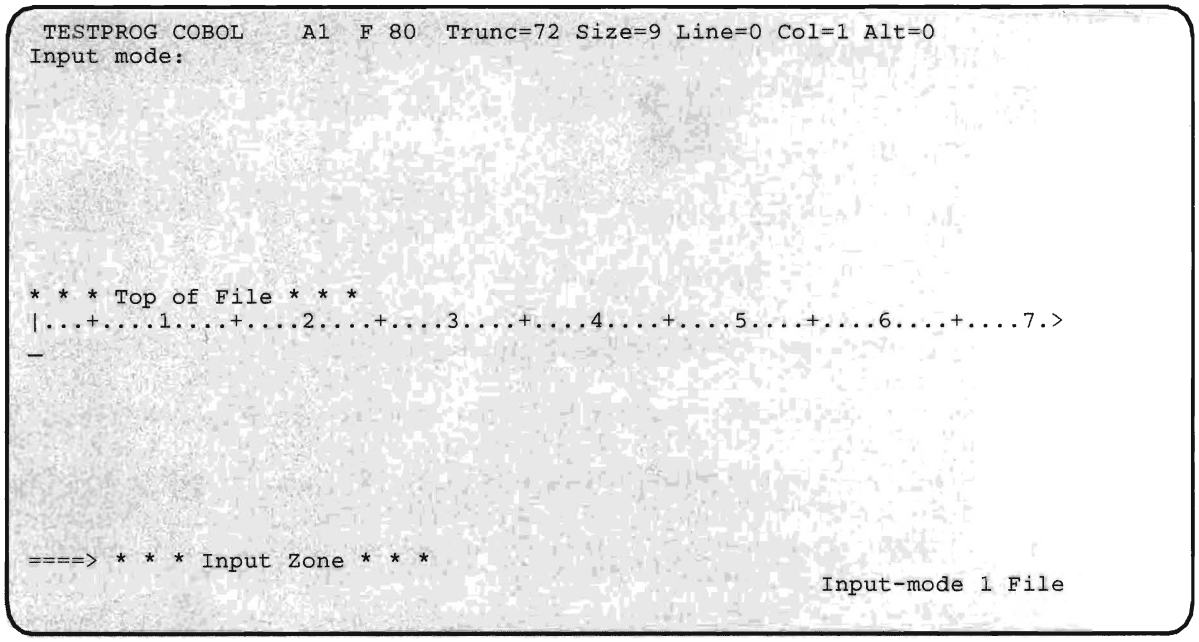
Alt=0	indicates the number of alterations that have been made to the file since the file has been saved.
=====	is the prefix area of each display line.
====>	points to the command area. Next to the arrow is the **cursor**. The cursor indicates where the next character you key in will appear.
X E D I T 1 File	is the session identifier message. It indicates the number of files you're editing.

The editor operates in two basic modes: the **edit** mode and the **input** mode. We'll use the **input mode** of the editor to create, and add to, the file. For more information about the editor, see "Chapter 3: Using the System Product Editor" on page 39.

To get into the input mode, enter:

```
i
```

to invoke the editor and receive the initial input mode display:



```
 TESTPROG COBOL    A1  F 80  Trunc=72 Size=9 Line=0 Col=1 Alt=0
Input mode:








* * * Top of File * * *
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
_








====> * * * Input Zone * * *
                                                       Input-mode 1 File
```

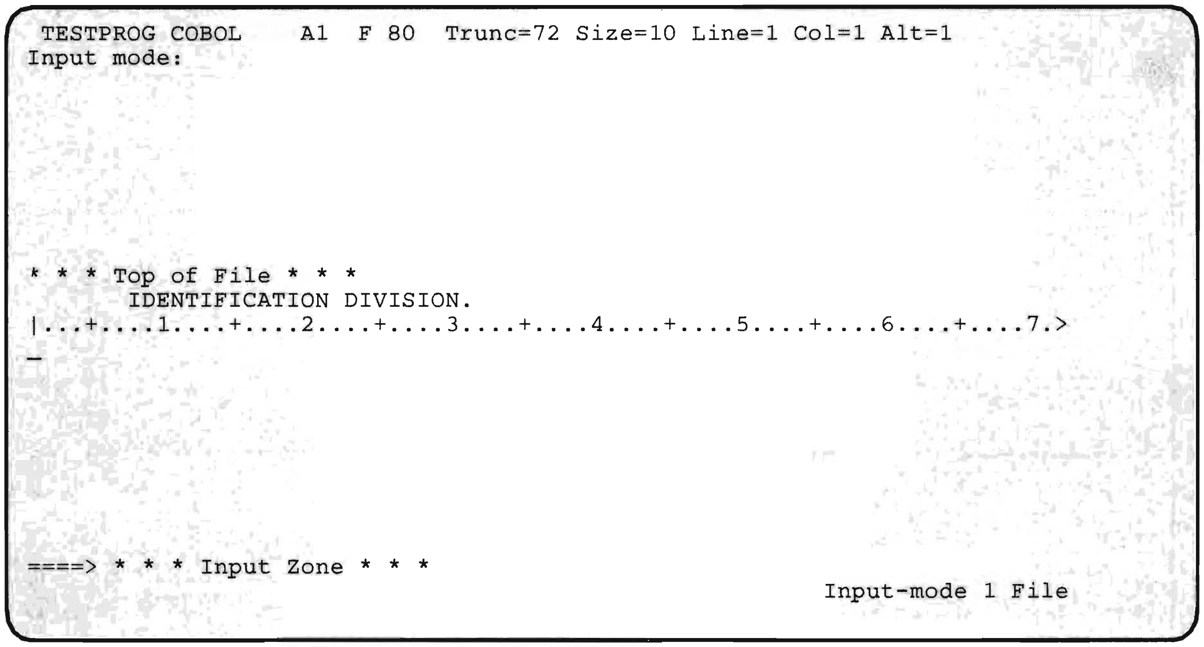
The cursor is now positioned at the first character of the first line. You're ready to enter a source program into the file. (See Appendix A, "Complete COBOL Program Examples" on page 271 for a complete listing of this program.) Press **PF4** or **PF16** to tab over to column 8. If you don't have PF keys on your keyboard, space over to column 8.

Enter:

```
identification division.
```

The cursor is now at the first data position of the second line.

Your display screen looks like this:

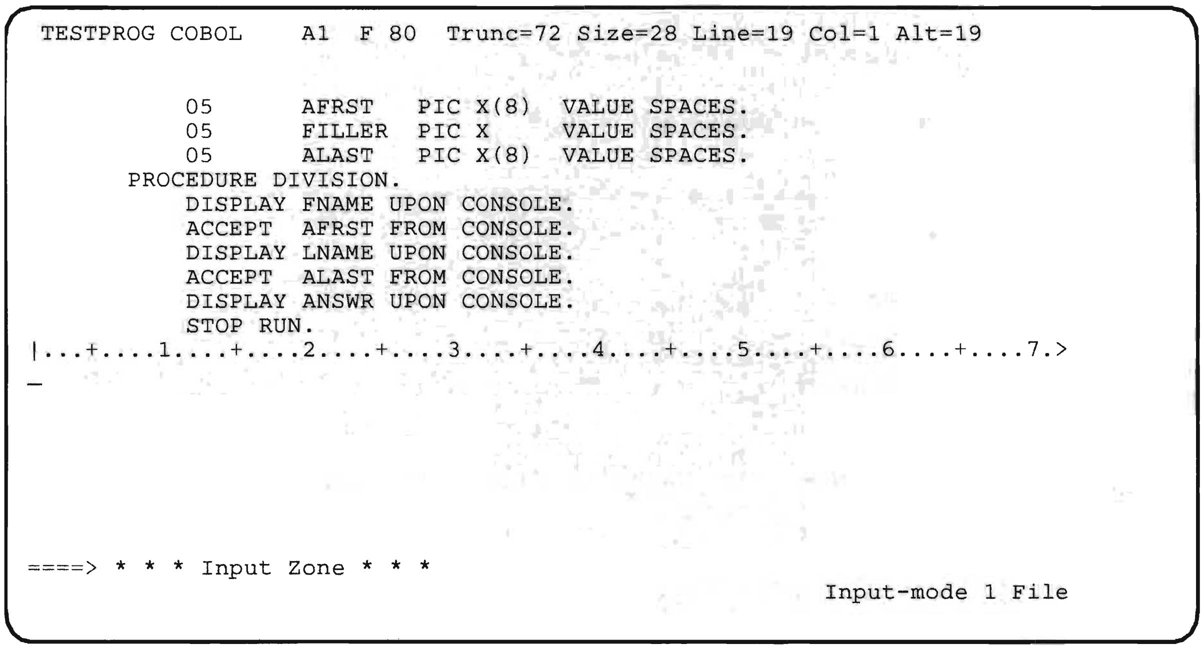


```
 TESTPROG COBOL     A1  F 80  Trunc=72 Size=10 Line=1 Col=1 Alt=1
Input mode:







* * * Top of File * * *
       IDENTIFICATION DIVISION.
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
_







====> * * * Input Zone * * *
                                                       Input-mode 1 File
```

Continue entering the following lines in this manner until the full program has been entered. To enter several lines at a time, use the RETURN key rather than the ENTER key. You don't have to press the ENTER key until you've filled up the input area.

```
PROGRAM-ID. MYPROG.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  FNAME          PIC X(22) VALUE "ENTER YOUR FIRST NAME.".
77  LNAME          PIC X(23) VALUE "AND NOW YOUR LAST NAME.".
01  ANSWR.
    05    ANSLT    PIC X(16) VALUE "WELCOME TO CMS, ".
    05    AFRST    PIC X(8)  VALUE SPACES.
    05    FILLER   PIC X     VALUE SPACES.
    05    ALAST    PIC X(8)  VALUE SPACES.
PROCEDURE DIVISION.
    DISPLAY FNAME UPON CONSOLE.
    ACCEPT  AFRST FROM CONSOLE.
    DISPLAY LNAME UPON CONSOLE.
    ACCEPT  ALAST FROM CONSOLE.
    DISPLAY ANSWR UPON CONSOLE.
    STOP RUN.
```

Your display now looks like this:



```
 TESTPROG COBOL     A1  F 80  Trunc=72 Size=28 Line=19 Col=1 Alt=19


           05      AFRST   PIC X(8)  VALUE SPACES.
           05      FILLER  PIC X     VALUE SPACES.
           05      ALAST   PIC X(8)  VALUE SPACES.
       PROCEDURE DIVISION.
           DISPLAY FNAME UPON CONSOLE.
           ACCEPT  AFRST FROM CONSOLE.
           DISPLAY LNAME UPON CONSOLE.
           ACCEPT  ALAST FROM CONSOLE.
           DISPLAY ANSWR UPON CONSOLE.
           STOP RUN.
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
_






====> * * * Input Zone * * *
                                                     Input-mode 1 File
```

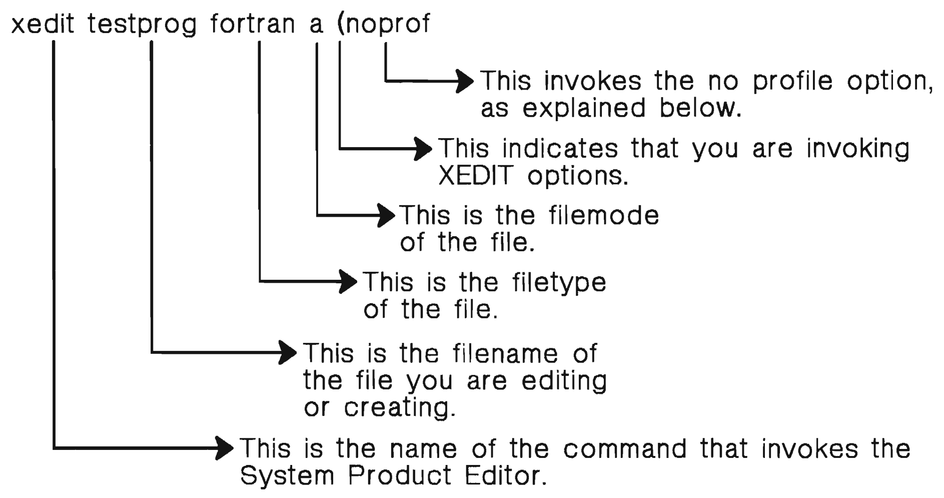
Now press ENTER to put the cursor on the command line.

If you are programming in COBOL, skip to "Saving Your File" on page 29.

Creating a FORTRAN File

You invoke the System Product Editor by entering the **XEDIT** command, followed by the file identifier of the file you want to create or edit. If there are options you want to apply to this session, put them next, after an open parenthesis.

For example, your **XEDIT** command might look like this:



```
xedit testprog fortran a (noprof
```

This invokes the no profile option, as explained below.

This indicates that you are invoking XEDIT options.

This is the filemode of the file.

This is the filetype of the file.

This is the filename of the file you are editing or creating.

This is the name of the command that invokes the System Product Editor.

Note: If you plan to go through this book doing both the COBOL and FORTRAN examples, make sure you give the two files you create different filenames (that is, you must change **testprog** to something else for the second language).

In our example we'll use a file with the **file identifier** TESTPROG FORTRAN A. The **filename** (TESTPROG), and the **filetype** (FORTRAN), are parts of the identifier assigned by you when you create the file. The **filemode** designates on which disk the file is to reside. CMS assigns A if the filemode is not specified for a new file.

Issue the **XEDIT** command now. Enter:

```
xedit testprog fortran a (noprof
```

(You can use **X** as an abbreviation for **XEDIT**.)

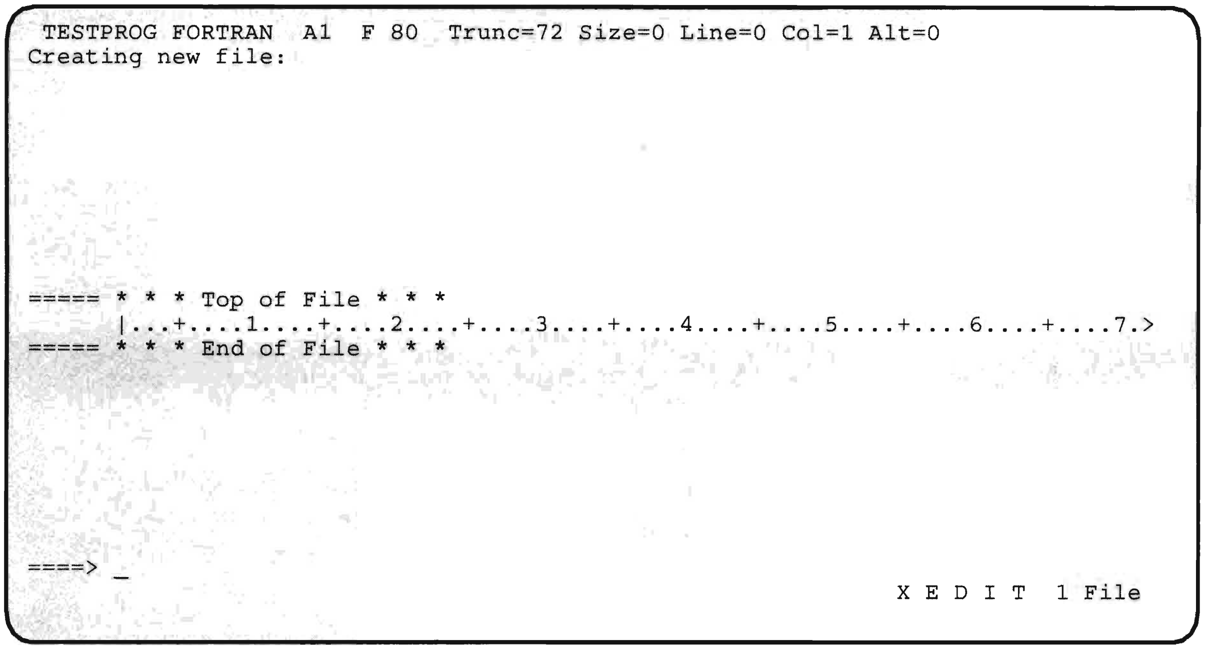
We're using the **noprof** (no profile) option to make sure that your screen looks as shown in this manual, no matter how your installation has customized the programmer environment. Normally you'd enter the command:

```
xedit testprog fortran a
```

or just

```
x testprog fortran
```

Your display screen should look like this:



```
 TESTPROG FORTRAN  A1  F 80  Trunc=72 Size=0 Line=0 Col=1 Alt=0
Creating new file:








===== * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== * * * End of File * * *








====> _
                                                         X E D I T  1 File
```

Here are what the various parts of the screen mean:

TESTPROG	is the filename.
FORTRAN	is the filetype.
A1	is the filemode.
F	is the type of file format, where: • F = fixed length records • V = variable length records
80	is the record length.
Trunc=72	indicates when truncation begins. If the number is less than the record length, data entered at this column or beyond is ignored.
Size=0	is the number of records in the file.
Line=0	indicates the line number of the current line.
Col=1	indicates the position of the column pointer. The column pointer is the vertical bar (|) in column 1 of the scale.

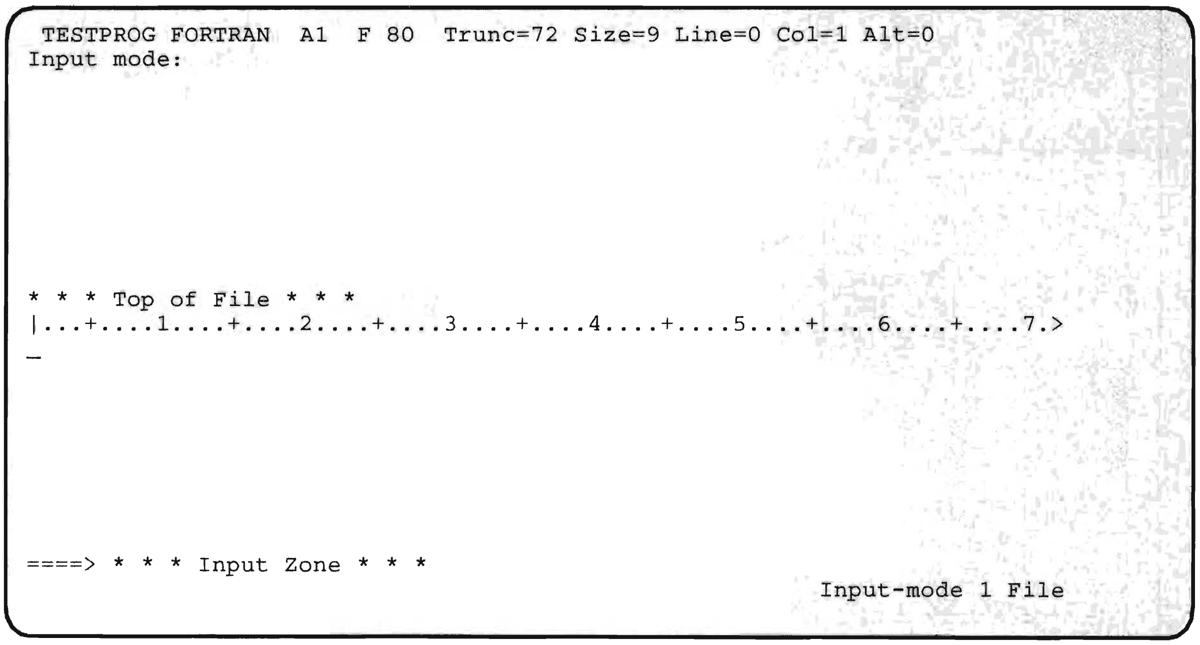
Alt=0	indicates the number of alterations that have been made to the file since the file has been saved.
=====	is the prefix area of each display line.
====>	points to the command area. Next to the arrow is the **cursor**. The cursor indicates where the next character you key in will appear.
X E D I T 1 File	is the session identifier message. It indicates the number of files you're editing.

The editor operates in two basic modes: the **edit** mode and the **input** mode. We'll use the **input mode** of the editor to create, and add to, the file. For more information about the editor, see "Chapter 3: Using the System Product Editor" on page 39.

To get into the input mode, enter:

```
i
```

to invoke the editor and receive the initial input mode display:



```
 TESTPROG FORTRAN  A1  F 80  Trunc=72 Size=9 Line=0 Col=1 Alt=0
Input mode:








* * * Top of File * * *
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
_







====> * * * Input Zone * * *
                                                   Input-mode 1 File
```

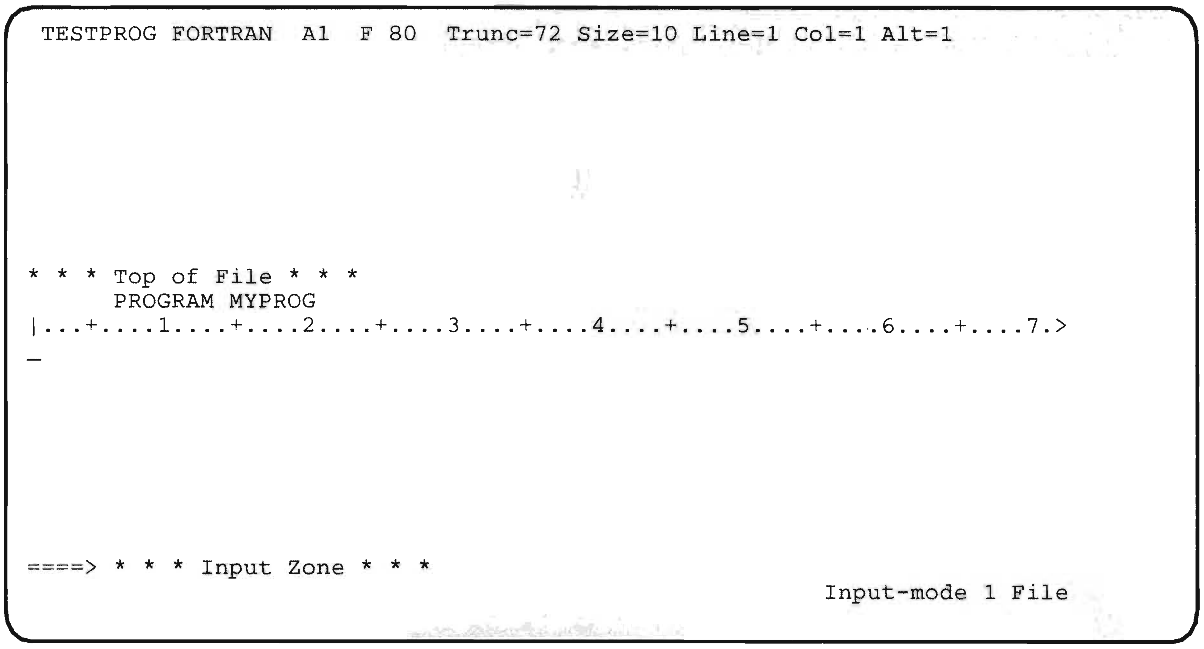
The cursor is now positioned at the first character of the first line. You're ready to enter a source program into the file. (See Appendix B, "Complete FORTRAN Program Examples" on page 277 for a complete listing of this program.) Press **PF4** or **PF16** to tab over to column 7. If you don't have PF keys on your keyboard, space over to column 7.

Enter:

```
program myprog
```

The cursor is now at the first data position of the second line.

Your display screen looks like this:

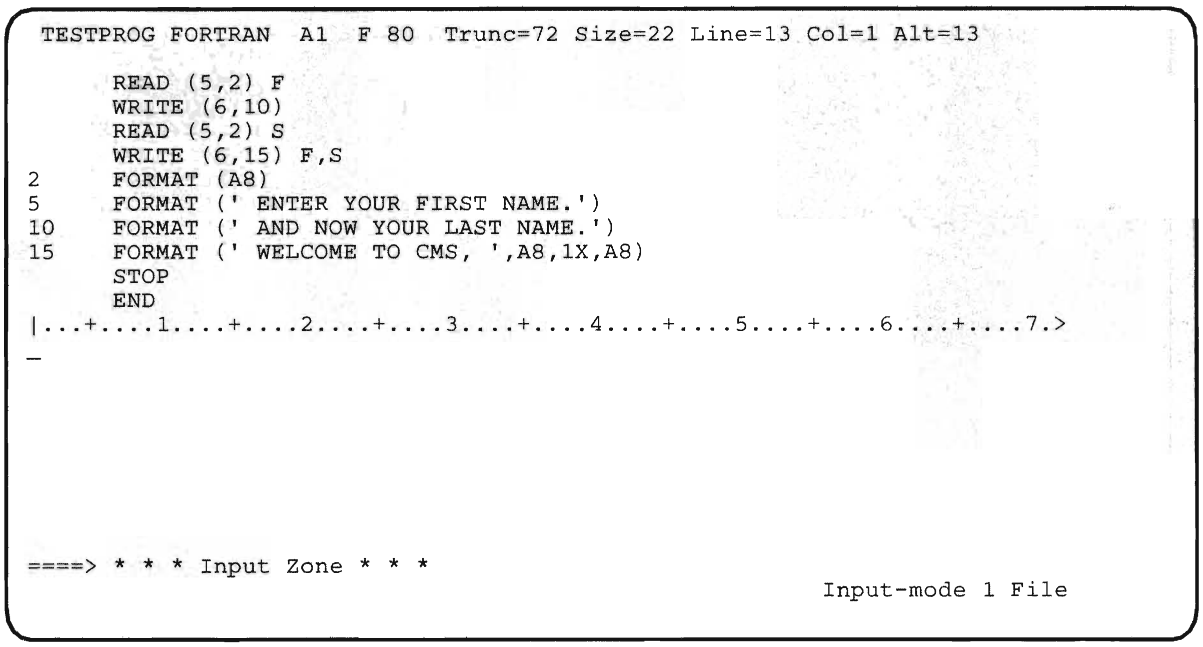


```
 TESTPROG FORTRAN  A1  F 80  Trunc=72 Size=10 Line=1 Col=1 Alt=1






* * * Top of File * * *
      PROGRAM MYPROG
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
_






====> * * * Input Zone * * *
                                                        Input-mode 1 File
```

Continue entering the following lines in this manner until the complete program has been entered. To enter several lines at a time, use the RETURN key (<--) rather than the ENTER key. You don't have to press the ENTER key until you've filled up the input area.

```
      CHARACTER*8 F,S
      WRITE (6,5)
      READ (5,2) F
      WRITE (6,10)
      READ (5,2) S
      WRITE (6,15) F,S
2     FORMAT (A8)
5     FORMAT (' ENTER YOUR FIRST NAME.')
10    FORMAT (' AND NOW YOUR LAST NAME.')
15    FORMAT (' WELCOME TO CMS, ',A8,1X,A8)
      STOP
      END
```

Your display now looks like this:



```
 TESTPROG FORTRAN  A1  F 80  Trunc=72 Size=22 Line=13 Col=1 Alt=13

      READ (5,2) F
      WRITE (6,10)
      READ (5,2) S
      WRITE (6,15) F,S
2     FORMAT (A8)
5     FORMAT (' ENTER YOUR FIRST NAME.')
10    FORMAT (' AND NOW YOUR LAST NAME.')
15    FORMAT (' WELCOME TO CMS, ',A8,1X,A8)
      STOP
      END
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
_






====> * * * Input Zone * * *
                                                  Input-mode 1 File
```

Now press ENTER to put the cursor on the command line.

Saving Your File

At this point, let's save the file. This can be done in one of two ways:

- **SAVE** and continue the session
- **FILE** and end the session

To do either you must leave the input mode and issue a command from the command line. To leave the input mode press the **ENTER** key twice, if you haven't already done so.

Now that the cursor is at the command line, enter:

```
save
```

The file is saved and we're still in our editing session. The screen has not been cleared and we can continue working.

It's a good idea to issue the **SAVE** command periodically while working on a file. This ensures that your work is not lost in the event of a system problem.

The **FILE** command saves the file, and also ends the session. Because you've just issued the **SAVE** command, you're no longer in the input mode and can issue the **FILE** command now. To see this, enter the **FILE** command (the cursor should still be in the command line):

```
file
```

The editor session is over, and the file has been saved. The editor screen no longer appears and the standard CMS ready message is displayed.

Step 2: Compile The Program

We're now going to compile the program you've written. To compile a program, you need:

- Sufficient storage.
- The appropriate CMS command for the compiler you want to use (COBOL or FORTVS2).
- The filename of the source file. You created the source program using the editor.
- Any compiler options that you want. These options are equivalent to the options you'd specify on the PARM parameter of an EXEC job control statement.

If you're compiling in COBOL, you probably have enough storage for the compiler. If you're using the VS FORTRAN Version 2 compiler, you'll need at least 3M to compile your program.

To find out how much storage you have, enter:

```
query storage
```

The system responds with a message like:

```
STORAGE = 512K (or 1024K or 2048K or 3060K)
```

Get more storage, by entering:

```
define storage 2048K
```

or

```
define storage 2M
```

Then type IPL CMS, and press enter twice.

After invoking the compiler and after the compiler finishes executing, it displays compilation error messages on your terminal, and writes them into the LISTING file.

The diagnostic messages displayed during compilation help you identify problems in the source program. You may not have to examine the LISTING file at all.

If necessary, you can review the LISTING file by using the editor. The editor lets you edit multiple files and display multiple screen images. You can display both the source program and the listing at the same time (the source program on the top half of the screen, the listing on the bottom half.). If your diagnostic messages indicate errors in the source program, you can examine the messages in the listing and correct the corresponding source statements during the same editor session.

To print a copy of the listing on your virtual printer, you use the **PRINT** command. For some installations, you may have to also transfer from your virtual printer to the specific printer that you want your output to be printed on.

Along with diagnostic messages and the LISTING file, the compiler produces a relocatable object module with a filetype of **TEXT**. You can load the TEXT file into storage when you want to execute your program.

If you're programming in FORTRAN, skip to "Compiling Your FORTRAN Program" on page 34.

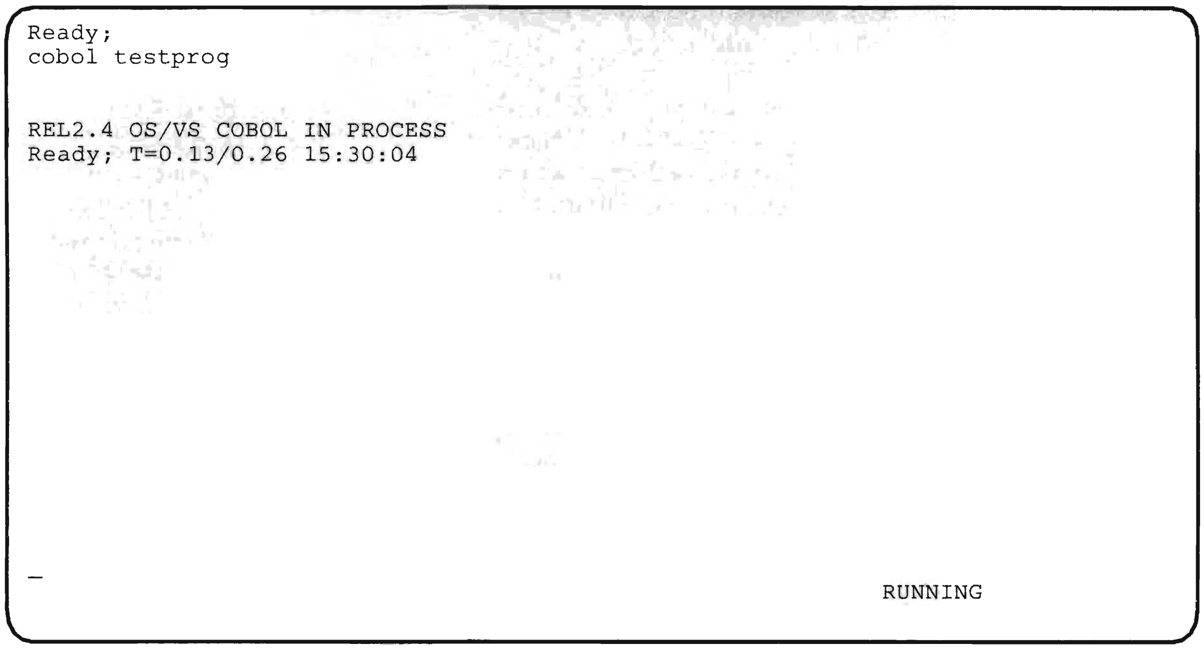
Compiling Your COBOL Program

The CMS command that invokes the OS/VS COBOL Compiler is simply **COBOL**.

Enter the following line on your terminal:

```
cobol testprog
```

The OS/VS COBOL compiler is now compiling your program. When the program has compiled correctly, it displays a message:



```
Ready;
cobol testprog


REL2.4 OS/VS COBOL IN PROCESS
Ready; T=0.13/0.26 15:30:04












_
                                                            RUNNING
```

If you had any errors in your program:

- Edit the source program.
- Correct the errors and enter a FILE command.
- Compile the program again.

When the compiler is finished, enter:

```
filelist testprog
```

This lists all the files on your A-disk with the filename TESTPROG.

The TEXT file created by the COBOL compiler contains the machine-language object code. This file is only executable in the CMS environment. (See "Step 3: Run The Program" on page 37.) The LISTING file contains the compilation listing for the source file you're compiling.

To print a copy of the LISTING file, use the **PRINT** command: (For some installations, you may have to also transfer from your virtual printer to the specific printer that you want your output to be printed on.)

```
print testprog listing
```

Figure 5 on page 33 illustrates what happens in your system when you compile your program.

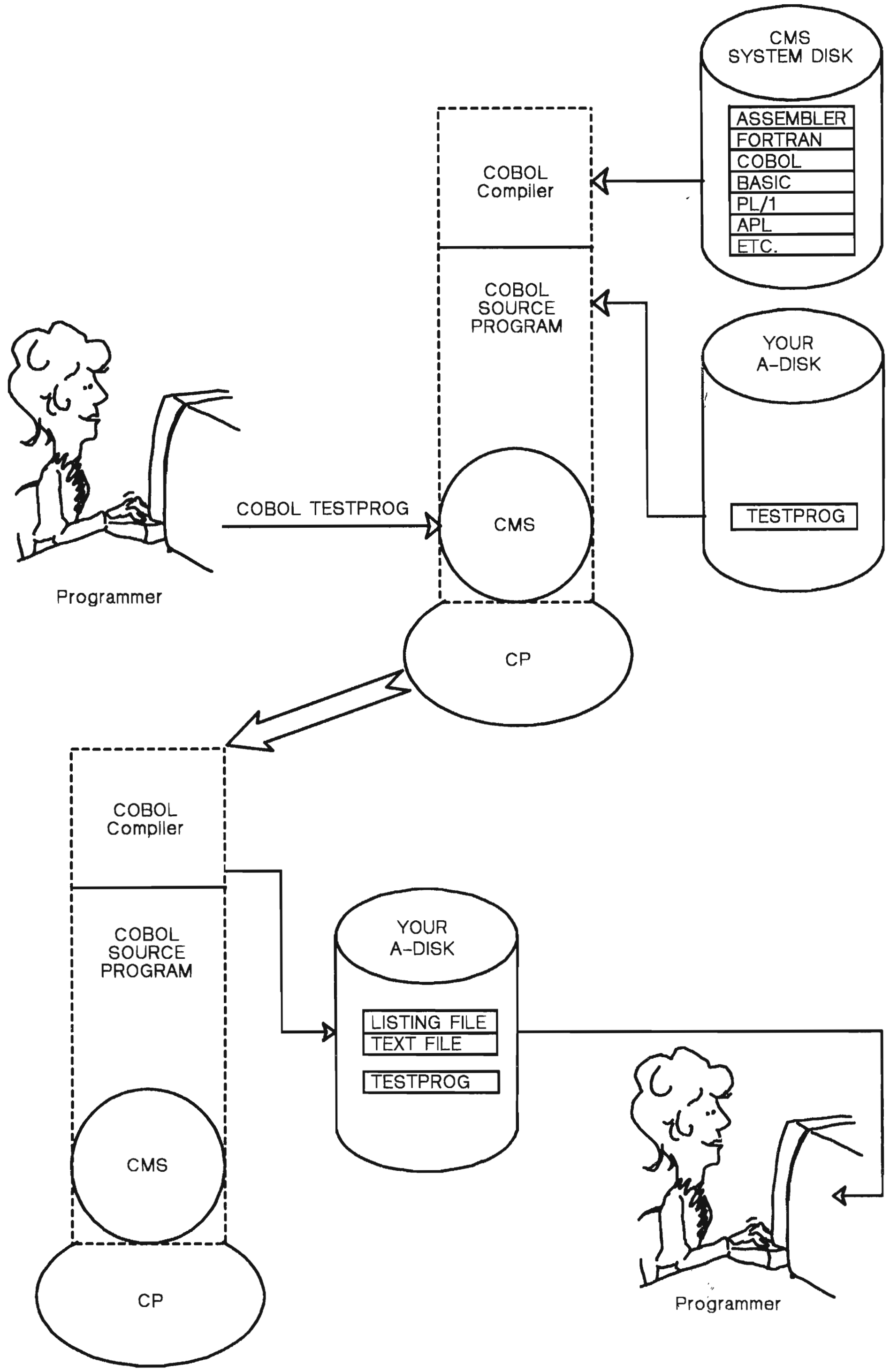


Figure 5. Compiling a COBOL Program

If you're programming in COBOL, skip to "Step 3: Run The Program" on page 37.

Compiling Your FORTRAN Program

Several FORTRAN compilers are available to you. We'll use the VS FORTRAN Version 2 compiler. The file must contain 80-byte, fixed-length records. The filetype must be FORTRAN.

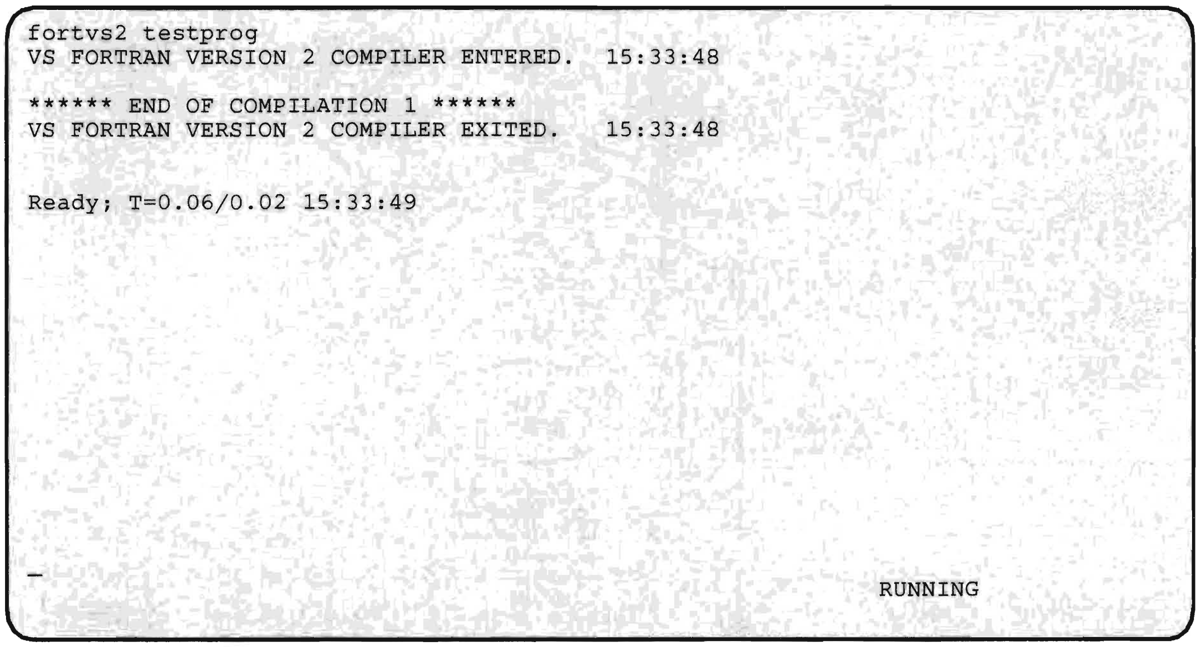
Enter the following line:

```
fortvs2 testprog
```

The compiler is now compiling your program.

Figure 6 on page 36 shows what happens in the VM system when you compile a FORTRAN program.

When the program has compiled correctly, you will see the following message on your screen:



```
fortvs2 testprog
VS FORTRAN VERSION 2 COMPILER ENTERED.   15:33:48

****** END OF COMPILATION 1 ******
VS FORTRAN VERSION 2 COMPILER EXITED.    15:33:48


Ready; T=0.06/0.02 15:33:49









_                                                              RUNNING
```

If you had any errors in your program:

- Edit the source program.
- Correct the errors and enter a FILE command.
- Compile the program again.

When the compiler is finished, enter:

```
filelist testprog
```

This lists all the files on your A-disk with the filename TESTPROG.

The TEXT file created by the FORTRAN compiler contains the machine-language object code. This file is executable in the OS and CMS environment. The entry point name for a main program in a TEXT file is the name you specified for the NAME option of the compiler command or its default name, MAIN. Subprograms have the entry point name that you specified in the FORTRAN SUBROUTINE statement.

The copy of the TEXT file pseudo-assembler listing included in your LISTING file contains an identification for the programs in it. Columns 73-76 of each line of code contain four characters that identify whether that code was generated for a main program or subprogram.

For main programs, the first four characters of the name specified in the PROGRAM statement, or the letters **MAIN** appear in columns 73-76. For subprograms, these columns contain the first four characters of the name specified in the SUBROUTINE statement. You can load and execute the TEXT file by issuing the **LOAD** command (see “Step 3: Run The Program” on page 37).

The LISTING file contains the compilation listing for the source file you're compiling.

To print a copy of the LISTING file, use the **PRINT** command: (For some installations, you may have to also transfer from your virtual printer to the specific printer that you want your output to be printed on.)

```
print testprog listing
```

Figure 6 on page 36 illustrates what happens in your system when you compile your program.

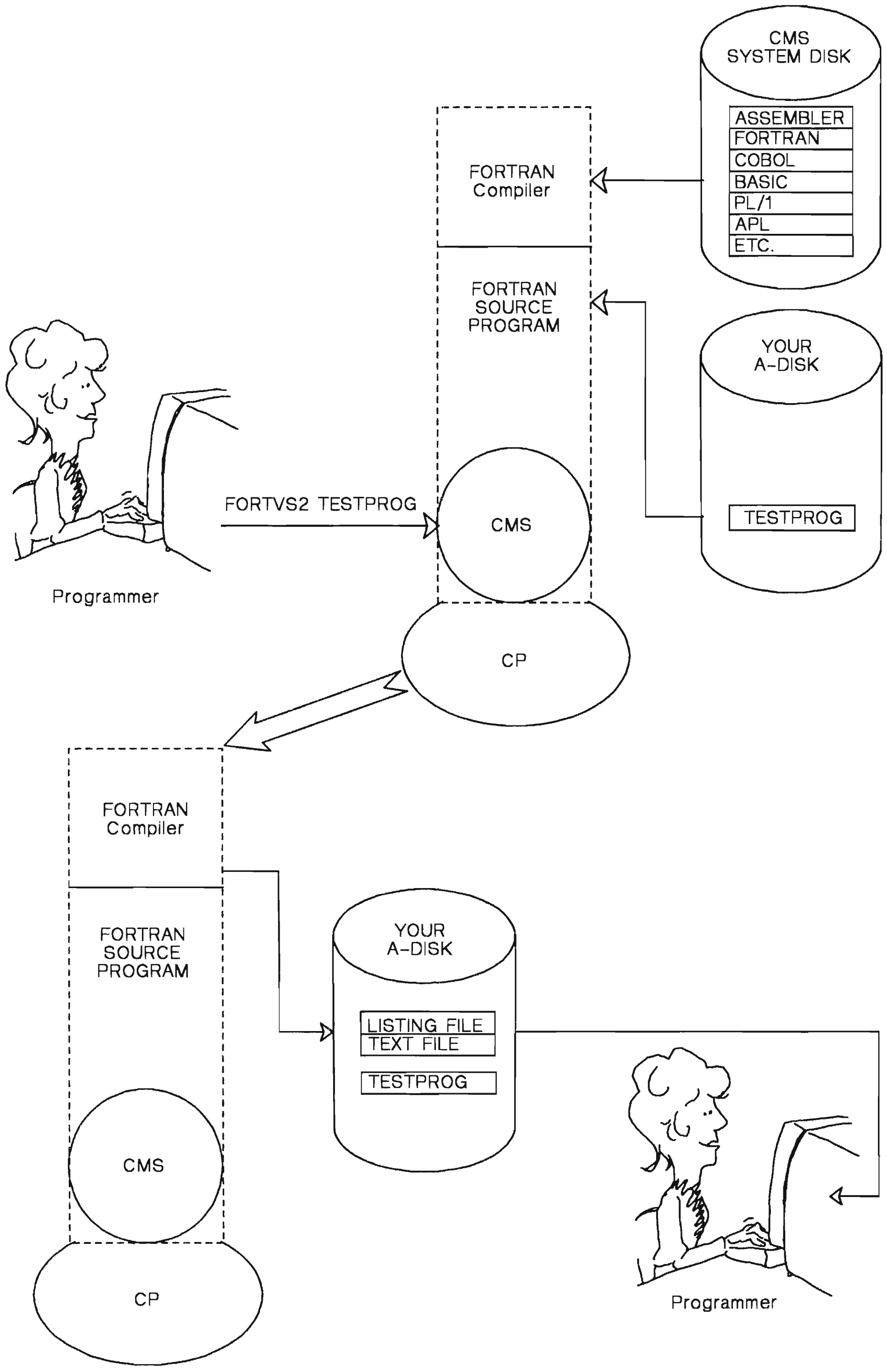


Figure 6. Compiling a FORTRAN Program

Step 3: Run The Program

Before you can run your program, you must tell VM the name of certain libraries. You can do this with the **GLOBAL** command. The exact **GLOBAL** commands you need depends on your installation. Ask your supervisor or system administrator for the exact **GLOBAL** command for your language and your installation.

Note: The order in which you specify library names in a **GLOBAL** command is very important. Be sure to use the exact order of names given to you.

A **GLOBAL** command to do this for COBOL programs might be:

```
global txtlib coblibvs
```

GLOBAL commands to do this for FORTRAN programs might be:

```
global txtlib vsf2fort cmslib
global loadlib vsf2load
```

After you've compiled your program, you can execute the TEXT files that were produced by the compiler. The TEXT files produced by the COBOL and FORTRAN compilers are relocatable and can be executed simply by loading them into virtual storage.

If you compiled your TESTPROG program with one of the COBOL or FORTRAN compilers mentioned in Step 2, you'll have a file on one of your disks called TESTPROG TEXT.

Since TESTPROG is a simple program that doesn't call subprograms or need to be linked with other modules, you can load and start the program with one command. You use the **LOAD** command with the **START** option:

```
load testprog (start
```

You also can do this in two steps. First, you issue the command:

```
load testprog
```

Then, to run the program, you issue the command:

```
start
```

Use one of these two methods to start the program. It displays the prompt:

```
ENTER YOUR FIRST NAME.
```

For this example, enter:

```
lee
```

The program responds by displaying the prompt:

```
AND NOW YOUR LAST NAME.
```

Enter:

```
green
```

The program responds by displaying:

```
WELCOME TO CMS, LEE   GREEN
```

The program has completed processing. If you like, you can run it again (with your own name, this time).

If You Have Problems...

If you're using this manual as a guide to creating programs, you might run into various problems you won't encounter with our sample programs. Here are solutions to two problem situations:

How to Get Out of an Infinite Loop

If this happens, you can use the **HX** command to halt execution of the program that is looping. You may not be able to "break in" with the **HX** command if your screen fills up with messages (as part of the loop). In this case enter:

```
#cp ext
```

This should cause the words **VM READ** to appear in the screen status area. You can then enter **HX**. If this still doesn't work, contact your programming supervisor for additional help.

How to Get Back to CMS from CP

If you find yourself in CP, and you need to get back to CMS, just type:

```
ipl cms
```

and press enter twice. You'll see the Ready; which lets you know that you're back in CMS.

Summary

In this chapter, you learned how to create a file, use the System Product Editor to enter a COBOL or FORTRAN source program, save the program, compile the program, and run it.

Chapter 3: Using the System Product Editor

In this chapter we'll discuss, in some detail, how to use the System Product Editor. We'll discuss what a CMS File is, how to use the EDIT and INPUT mode of the editor, how to manipulate data and the display of data, and how to use the CMS Update Facility.

What Is a CMS File?

Your CMS file is a collection of data. Each file has a unique **identifier**. The file identifier consists of a **filename**, a **filetype**, and a **filemode**. For example, in the case of a file called TESTFILE COBOL A, the filename is TESTFILE, the filetype is COBOL, and the filemode is A.

When you specify certain filetypes (such as COBOL, FORTRAN, and FREEFORT), the editor sets up default characteristics for the file. You can override these defaults during the editor session, but you'll probably not want to do so. The parameters governing these defaults are:

Case	controls whether or not characters entered are translated into upper case. All program language filetypes are translated to uppercase.
Tabs	controls where characters are placed in a record if preceded by a tab character.
Trunc	controls the columns within which the editor accepts changes to records in the file. Changes made outside the zones are ignored.
Verify	controls the columns that the editor displays on the screen. This is sometimes shorter than the length of a record. For example, when space has been reserved for sequence numbers in column 73-80, only columns 1-72 are displayed.

If you're editing a COBOL file, the default characteristics are:

Parameter	COBOL Default
Tabs	1 8 12 20 28 36 44 68 72 80
Trunc	72
Serial	On 10 10
RECFM	F
Width	80
Zone	1 72

If you're editing a FORTRAN file, the default characteristics are:

Parameter	FORTRAN Default
Tabs	1 7 10 15 20 25 30 80
Trunc	72
Serial	On 10 10
RECFM	F
Width	80
Zone	1 72

If you're editing a FREEFORT file, the default characteristics are:

Parameter	FREEFORT Default
Tabs	9 15 18 23 28 33 38 81
Trunc	81
Serial	Off 10 10
RECFM	V
Width	81
Zone	9 81

For the default characteristics of other filetypes, see *VM/SP System Product Editor Command and Macro Reference.*

Using the Edit and Input Modes

The editor has two basic modes of operation:

- Edit Mode
- Input Mode.

When you issue the **XEDIT** command, you enter the **edit mode**. In this mode, the editor initializes the edit session by taking control of the display screen and displaying the contents of the file you are about to edit. If you are creating a new file, the editor simply displays an empty file, to which you can add new records. If the file already exists, you can add new records and change or delete existing records. You do this by using **editor subcommands**, which we will refer to simply as **subcommands**.

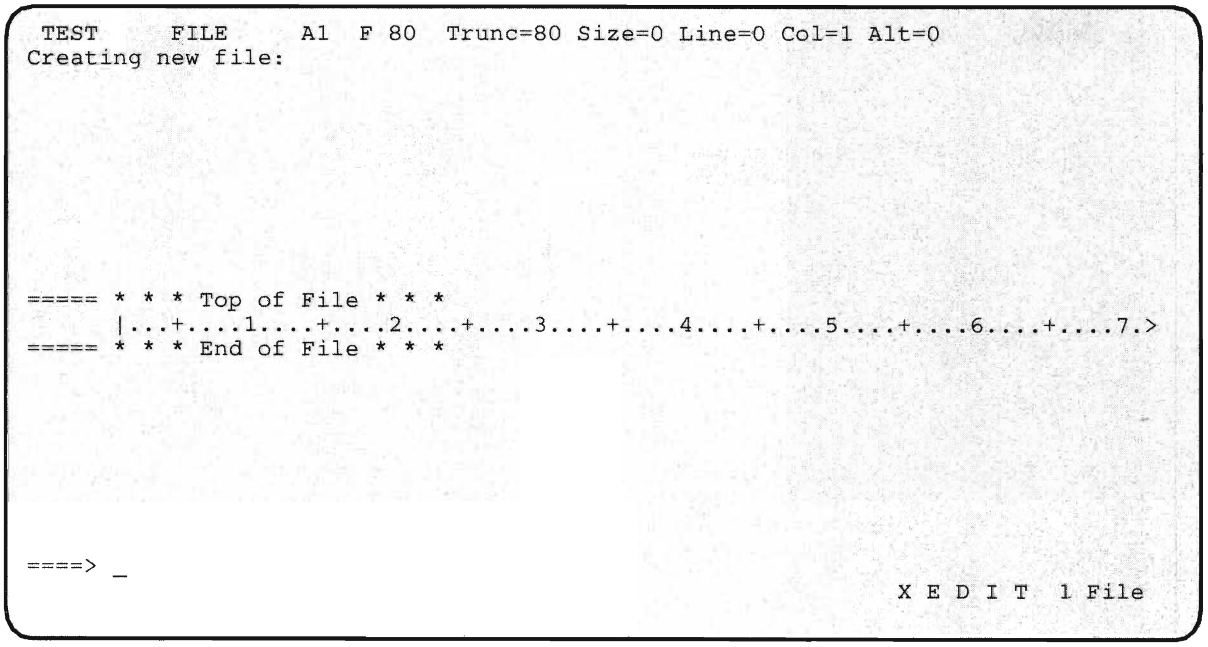
In “Step 1: Create a File” on page 19, you issued the **INPUT** subcommand, and then entered data to create a program. When you issue the **INPUT** subcommand (with no operands), you enter the **input mode**. (REPLACE and POWERINP also put the editor in the input mode; for information on these commands, see *VM/SP System Product Editor Command and Macro Reference.*)

Input is just one of many editor subcommands. You can use some of the editor subcommands for changing data in the file you are editing. Others you can use for changing the way the data is displayed on the screen. In this chapter, we'll examine some of the other subcommands and see how you can use them to ease the task of coding programs and data files.

Before we begin to use these commands, let's create a file to use just for demonstration. We'll call this TEST FILE A1. Enter the following line at the terminal:

```
xedit test file a (noprof
```

Since this file hasn't existed before, it's empty. Your screen should look like this:

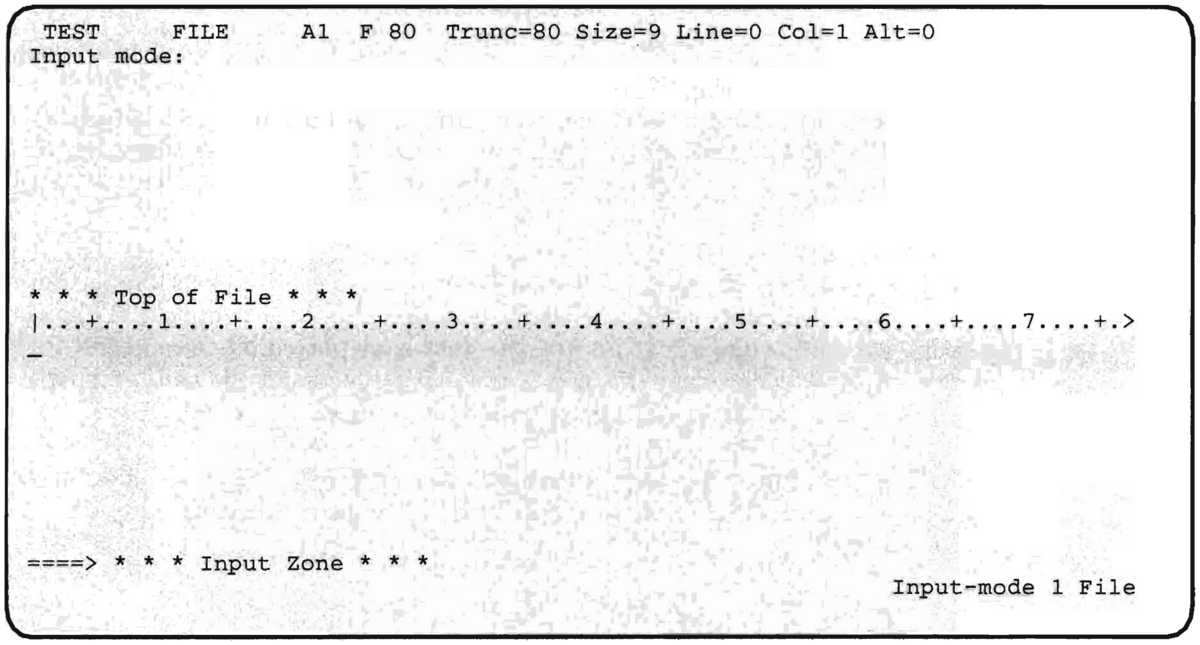


```
 TEST     FILE     A1  F 80  Trunc=80 Size=0 Line=0 Col=1 Alt=0
Creating new file:









===== * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== * * * End of File * * *









====> _
                                                       X E D I T  1 File
```

The cursor is on the command line. Enter:

```
input
```

Your screen should look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=9 Line=0 Col=1 Alt=0
Input mode:







* * * Top of File * * *
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+.>
_







====> * * * Input Zone * * *
                                                          Input-mode 1 File
```

Now enter the following lines of data (see note):

```
AAAAABBBBBCCCCC
BBBBBCCCCCDDDDD
CCCCCDDDDDEEEEE
DDDDDEEEEEFFFFF
EEEEEFFFFFGGGGG
FFFFFGGGGGHHHHH
GGGGGHHHHHIIIII
HHHHHIIIIIJJJJJ
IIIIIJJJJJKKKKK
JJJJJKKKKKLLLLL
KKKKKLLLLLMMMMM
LLLLLMMMMMNNNNN
MMMMMNNNNNOOOOO
NNNNNOOOOOPPPPP
OOOOOPPPPPQQQQQ
PPPPPQQQQQRRRRR
QQQQQRRRRRSSSSS
RRRRRSSSSSTTTTT
SSSSSTTTTTUUUUU
TTTTTUUUUUVVVVV
UUUUUVVVVVWWWWW
VVVVVWWWWWXXXXX
WWWWWXXXXXYYYYY
XXXXXYYYYYZZZZZ
YYYYYZZZZZAAAAA
ZZZZZAAAAABBBBB
```

Note: When you reach the bottom of the screen, press **PF8** or the enter key. Either will move the display FORWARD. Then you can enter the remaining lines.

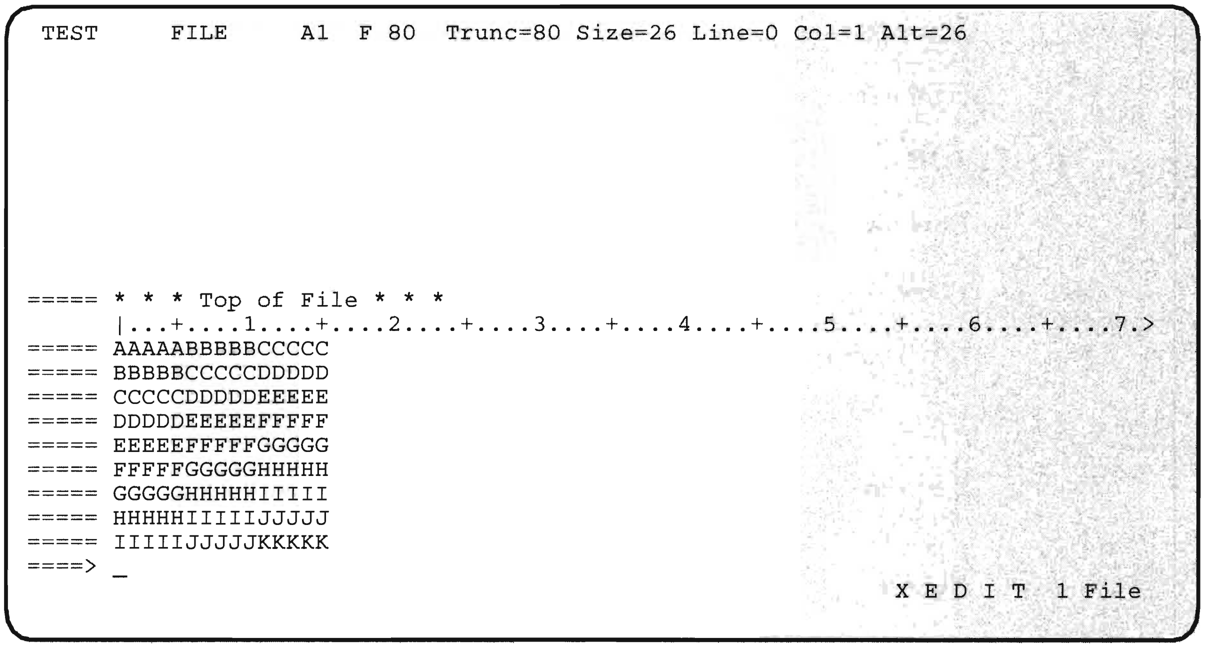
When you've entered the last line, press the ENTER key once more to leave the INPUT mode and return to the EDIT mode. The cursor automatically returns to the command line. Now enter the subcommand:

```
top
```

which changes the display so that the current line is at the top of the file, just before the first line of text.

In this case, the **current line** is the file line in the middle of the screen (above the scale). Most commands you type in the command line perform their functions starting with the current line. Naturally, the line that is current will change as you move up and down in the file.

Your screen should look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=26 Line=0 Col=1 Alt=26









===== * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== DDDDDEEEEEFFFFF
===== EEEEEFFFFFGGGGG
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
====> _
                                                  X E D I T  1 File
```

Some editor subcommands are invoked by typing them on the **command line**, which is the next to last line on your screen. It's easily identified by the character string ====>. This is displayed in high intensity on your terminal (if your terminal supports high and low intensity). When you begin an editor session, the cursor is two spaces beyond the command line symbol. This is the beginning of the command line itself.

In addition to the command line indicator, you will notice that each of the records you entered into the file has a string of five equal signs (=====) beside it. This is the prefix subcommand area. If you're using a full-screen terminal, you'll find this area more convenient, especially to manipulate data in a file.

Manipulating the Display

You can use some of the edit subcommands for manipulating the display. For example, in the previous section, you entered the subcommand **top** to change the display so that the current line was at the top of the file, just before the first line of text. Other subcommands let you move forward and backward in the file. That is, they let you move toward the last line (forward) or the first line (backward). Others allow you to move forward and back a specific number of lines.

Here are some of the command line subcommands you can use to manipulate the display:

TOP	Move line pointer to null Top of File line.
Bottom	Go to the last line of the file.
FOrward	Move the display one screen toward the bottom of the file.
BAckward	Move the display one screen toward the top of the file.
Up n	Move the line pointer n lines toward the top of the file.
Down n	Move the line pointer n lines toward the end of the file (same as NEXT)
Next n	Move the line pointer n lines toward the end of the file (same as DOWN)
RIght n	Move the display to the right **n** characters.
LEft n	Move the display to the left **n** characters.
Locate	Find a specified string of characters anywhere on a line.
Find	Find a specified string of characters at the beginning of a line.
:n	Make line **n** the current line.

Each of these subcommands affects the image of your file presented by the editor; the data in the file itself is not changed by them in any way.

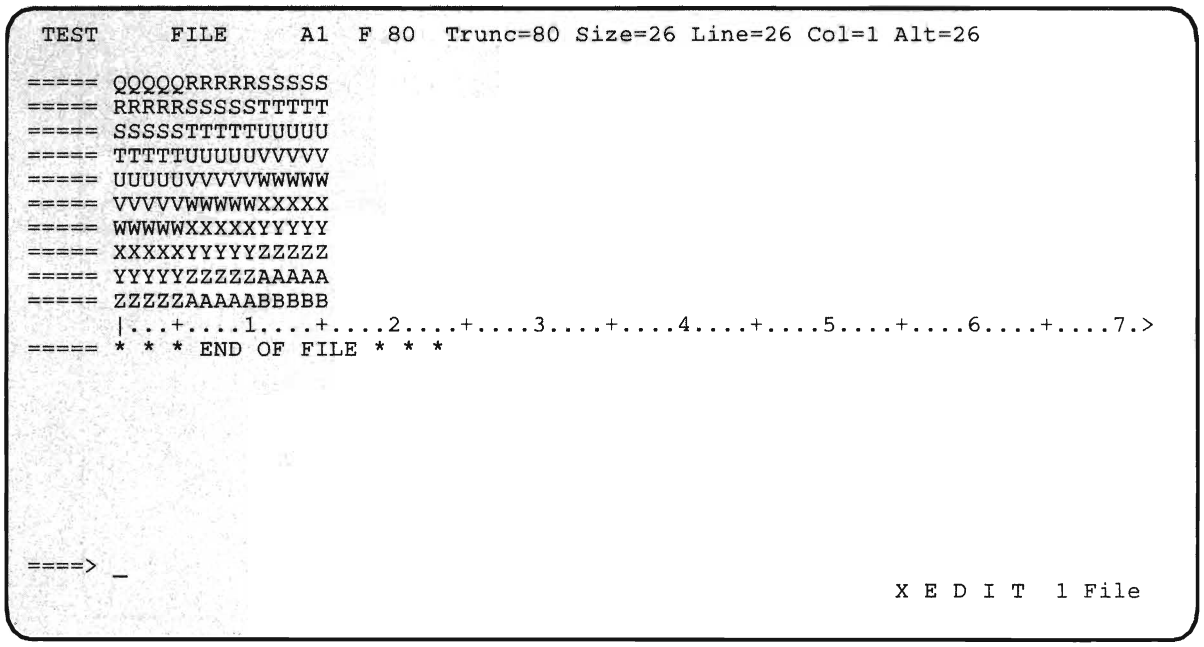
Examples

Let's see how the display changes when we use these subcommands. We've already used the **top** subcommand, and we saw that the display was changed so that the current line preceded the first line of the file. Now enter the subcommand

```
bottom
```

which changes the display so that the current line is the last line of the file.

Your screen should now look like this:



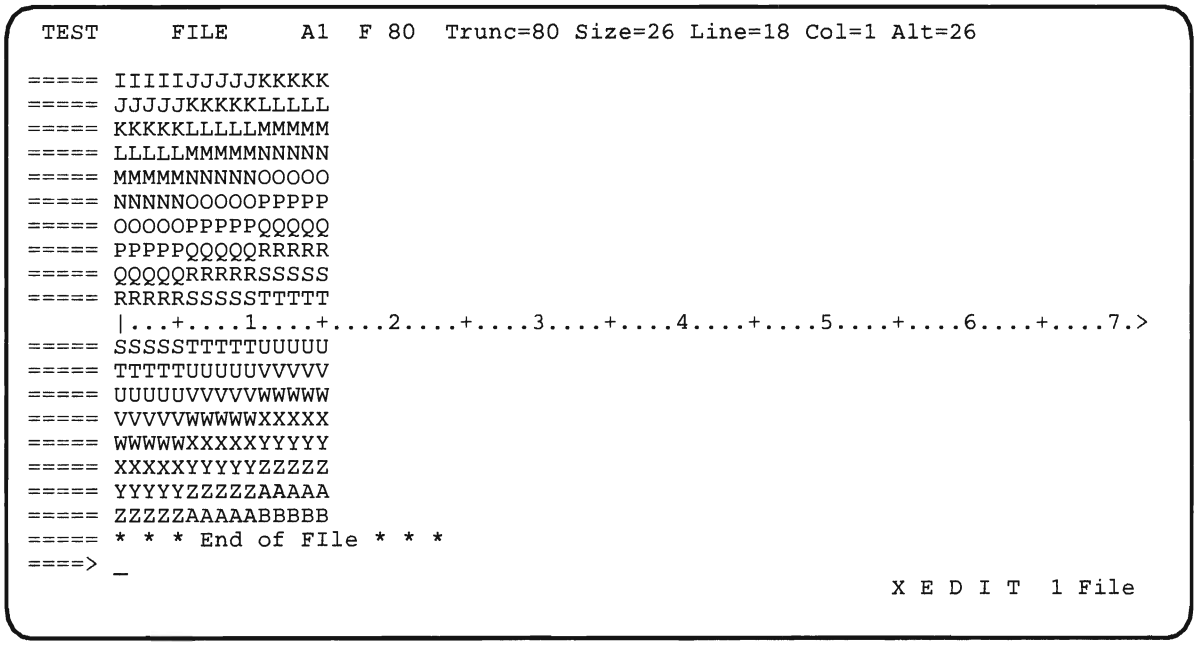
```
 TEST      FILE      A1  F 80  Trunc=80 Size=26 Line=26 Col=1 Alt=26

===== QQQQQRRRRRSSSSS
===== RRRRRSSSSSTTTTT
===== SSSSSTTTTTUUUUU
===== TTTTTUUUUUVVVVV
===== UUUUUVVVVVWWWWW
===== VVVVVWWWWWXXXXX
===== WWWWWXXXXXYYYYY
===== XXXXXYYYYYZZZZZ
===== YYYYYZZZZZAAAAA
===== ZZZZZAAAAABBBBB
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== * * * END OF FILE * * *







====> _
                                                                X E D I T  1 File
```

The subcommands **FORWARD** and **BACKWARD** are used to **scroll** through a file — that is, move toward the bottom or top of the file one screen at a time. **PF7** and **PF8** are preset by the editor to execute the **BACKWARD** and **FORWARD** commands. You can use these keys in place of the commands to scroll through your file. First, enter

```
top
```

to return to the top of the file. Now press **PF8**.

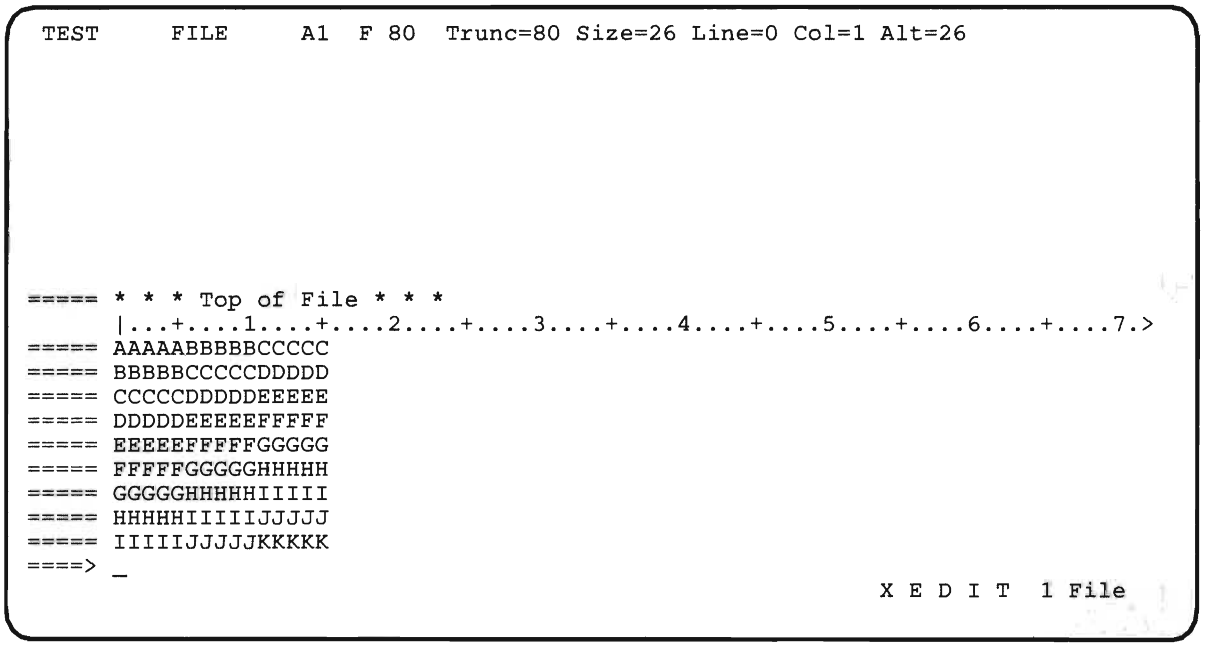
Your screen should now look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=26 Line=18 Col=1 Alt=26

===== IIIIIJJJJJKKKKK
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
===== OOOOOPPPPPQQQQQ
===== PPPPPQQQQQRRRRR
===== QQQQQRRRRRSSSSS
===== RRRRRSSSSSTTTTT
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== SSSSSTTTTTUUUUU
===== TTTTTUUUUUVVVVV
===== UUUUUVVVVVWWWWW
===== VVVVVWWWWWXXXXX
===== WWWWWXXXXXYYYYY
===== XXXXXYYYYYZZZZZ
===== YYYYYZZZZZAAAAA
===== ZZZZZAAAAABBBBB
===== * * * End of FIle * * *
====> _
                                                          X E D I T  1 File
```

Now press **PF7**.

Your screen should now look like this:



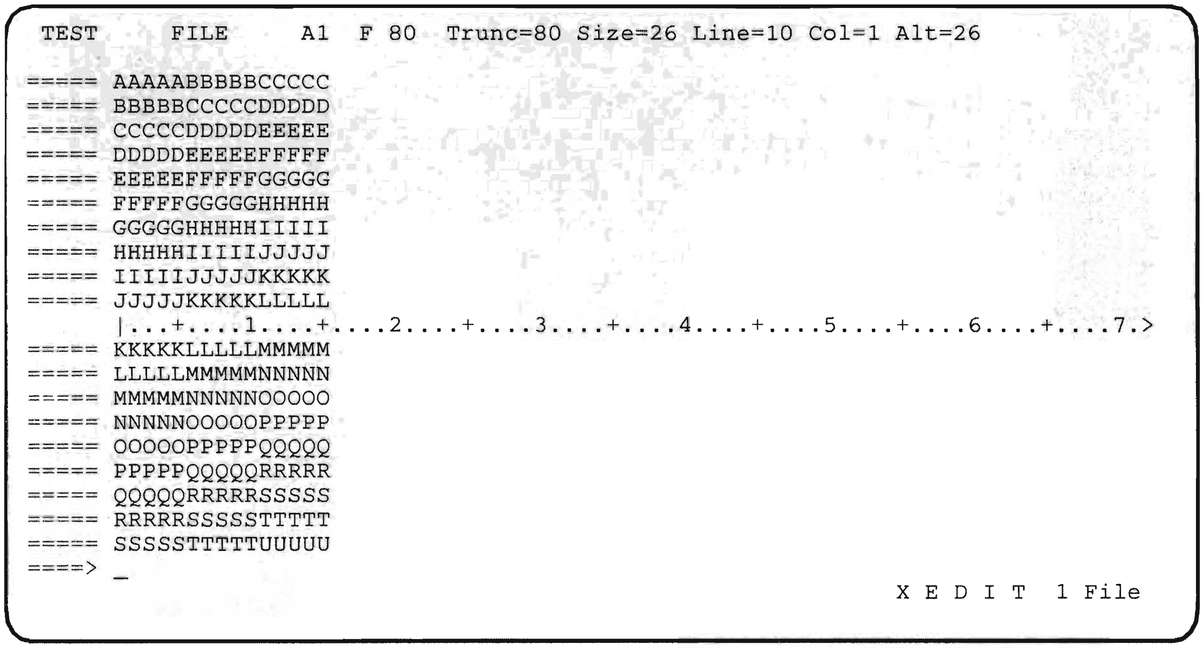
```
 TEST     FILE     A1  F 80  Trunc=80 Size=26 Line=0 Col=1 Alt=26









===== * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== DDDDDEEEEEFFFFF
===== EEEEEFFFFFGGGGG
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
====> _
                                                       X E D I T  1 File
```

If you want to move the display forward or backward less than a screen or more than a screen, you can use the **UP** and **DOWN** subcommands. (You can use the **NEXT** subcommand instead of **DOWN**.) For example, enter

```
down 10
```

to move the display 10 lines towards the bottom of the file.

Your screen should now look like this:



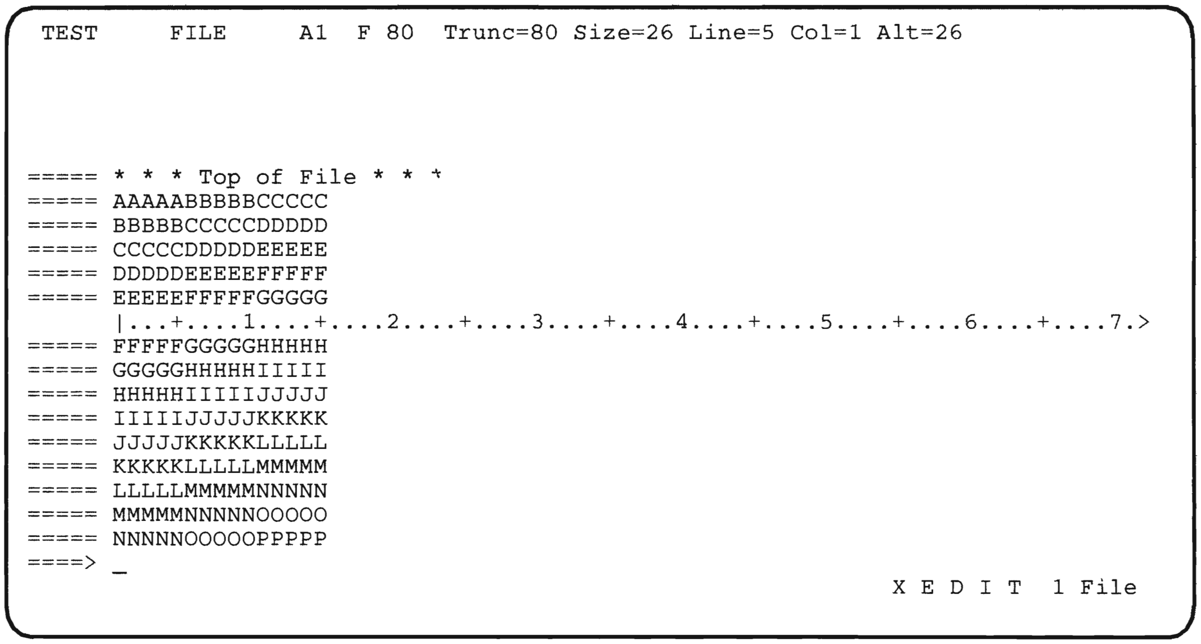
```
 TEST      FILE      A1  F 80  Trunc=80 Size=26 Line=10 Col=1 Alt=26

===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== DDDDDEEEEEFFFFF
===== EEEEEFFFFFGGGGG
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
===== JJJJJKKKKKLLLLL
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
===== OOOOOPPPPPQQQQQ
===== PPPPPQQQQQRRRRR
===== QQQQQRRRRRSSSSS
===== RRRRRSSSSSTTTTT
===== SSSSSTTTTTUUUUU
====> _
                                                      X E D I T  1 File
```

Another way to move the display down toward the bottom of the file is to simply use the number of lines, without the keyword **down**.

To move the display 5 lines toward the top of the file, enter

```
up 5
```

Your screen should now look like this:



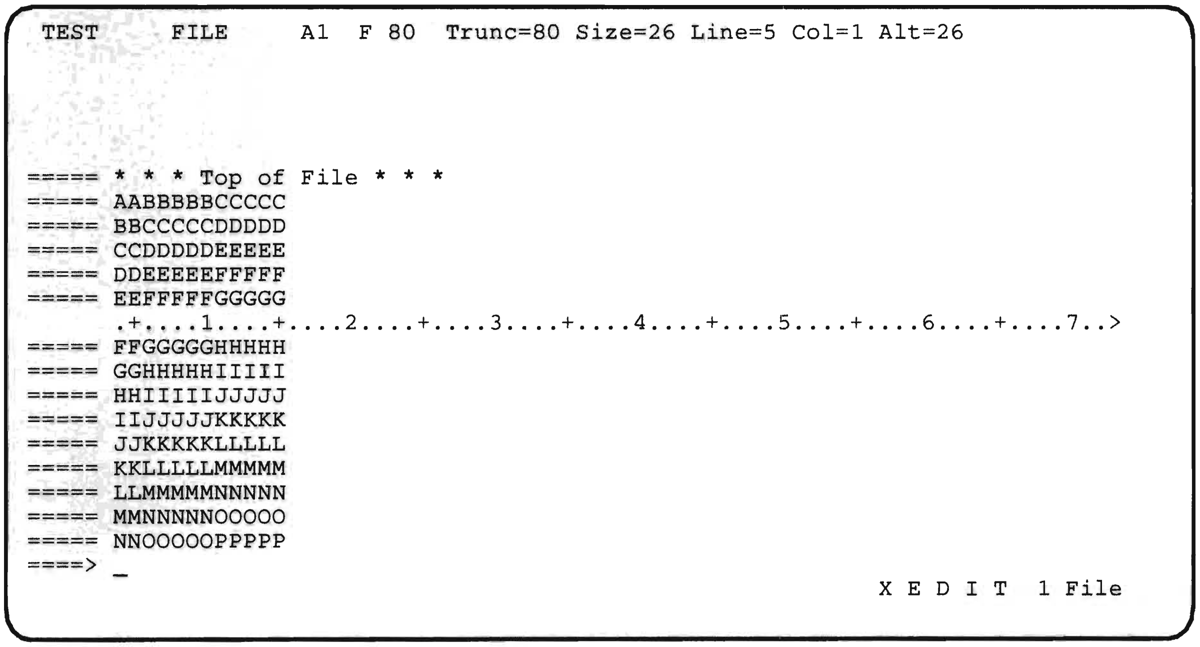
```
 TEST     FILE     A1  F 80  Trunc=80 Size=26 Line=5 Col=1 Alt=26




===== * * * Top of File * * *
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== DDDDDEEEEEFFFFF
===== EEEEEFFFFFGGGGG
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
====> _
                                                          X E D I T  1 File
```

Moving the display to the right or left **n** columns is called **horizontal scrolling**. For example, enter the subcommand:

```
right 3
```

to shift the display three characters to the right.

Your screen should now look like this:

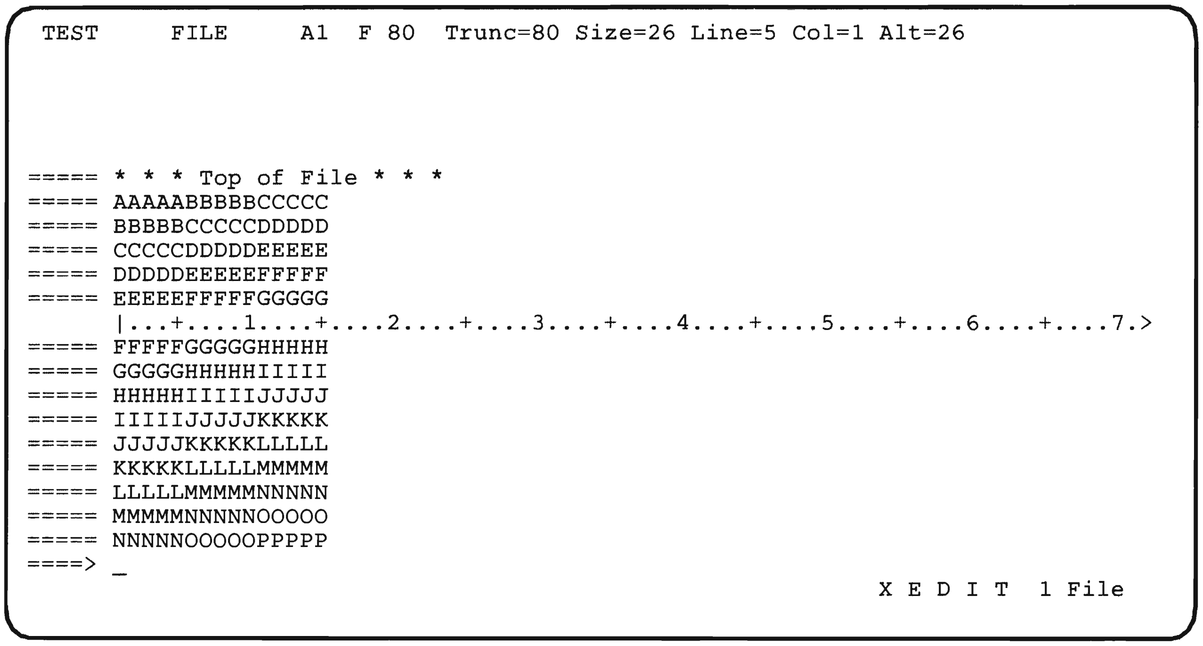


```
 TEST     FILE     A1  F 80  Trunc=80 Size=26 Line=5 Col=1 Alt=26




===== * * * Top of File * * *
===== AABBBBBCCCCC
===== BBCCCCCDDDDD
===== CCDDDDDEEEEE
===== DDEEEEEFFFFF
===== EEFFFFFGGGGG
      .+....1....+....2....+....3....+....4....+....5....+....6....+....7..>
===== FFGGGGGHHHHH
===== GGHHHHHIIIII
===== HHIIIIIJJJJJ
===== IIJJJJJKKKKK
===== JJKKKKKLLLLL
===== KKLLLLLMMMMM
===== LLMMMMMNNNNN
===== MMNNNNNOOOOO
===== NNOOOOOPPPPP
====> _
                                                          X E D I T  1 File
```

To move the display 3 characters to the left, enter:

```
left 3
```

Your screen should now look like this:

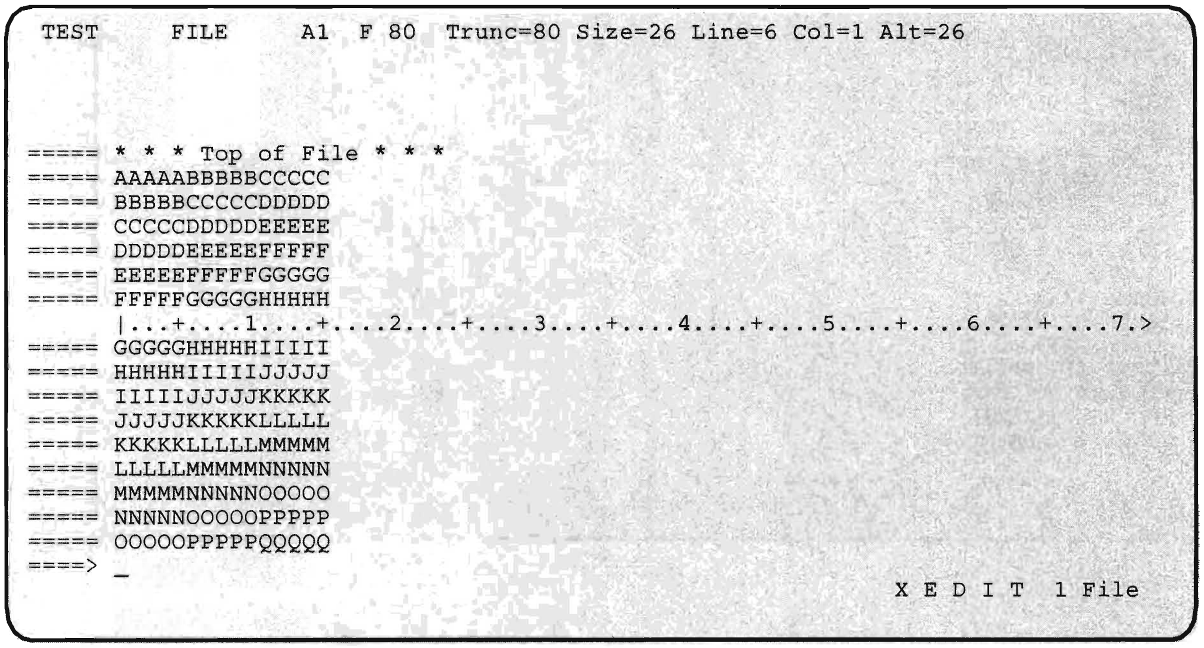


```
 TEST     FILE     A1  F 80  Trunc=80 Size=26 Line=5 Col=1 Alt=26




===== * * * Top of File * * *
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== DDDDDEEEEEFFFFF
===== EEEEEFFFFFGGGGG
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
====> _
                                                      X E D I T  1 File
```

To position the file display so that a particular line becomes the current line, you can use the **:n** subcommand. The **n** represents the number of the line relative to the beginning of the file. To make the sixth line the current line, enter

```
:6
```

Your screen should now look like this:



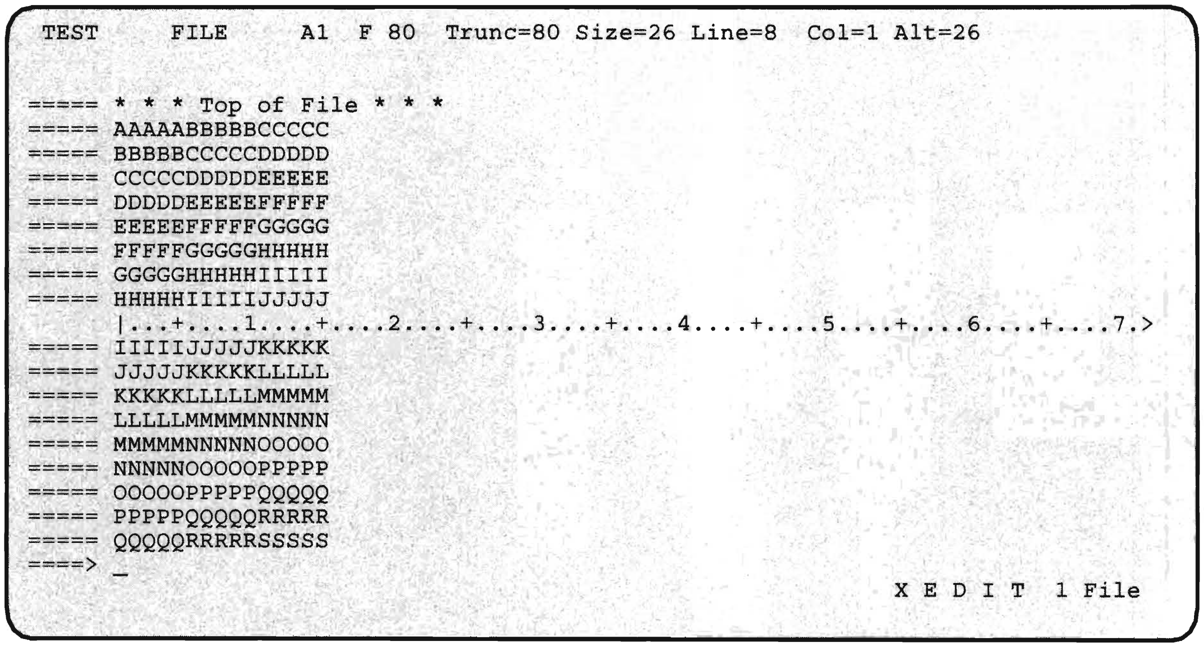
```
 TEST      FILE     A1  F 80  Trunc=80 Size=26 Line=6 Col=1 Alt=26



===== * * * Top of File * * *
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== DDDDDEEEEEFFFFF
===== EEEEEFFFFFGGGGG
===== FFFFFGGGGGHHHHH
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== GGGGGHHHHHIIIII
===== HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
===== OOOOOPPPPPQQQQQ
====> _
                                                          X E D I T  1 File
```

Use the **LOCATE** subcommand to locate a given character string. On the command line, enter:

```
locate/jjjjj/
```

The editor shows you the first line in which the string **jjjjj** appears.

Your screen should now look like this:



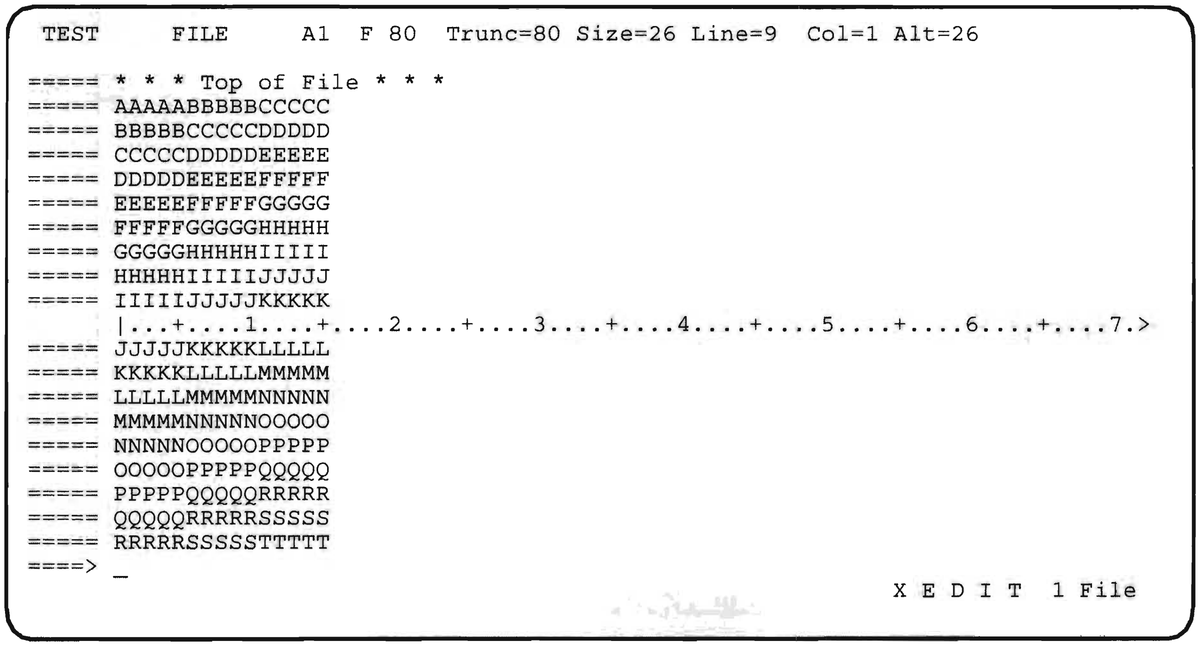
```
 TEST     FILE     A1  F 80  Trunc=80 Size=26 Line=8  Col=1 Alt=26


===== * * * Top of File * * *
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== DDDDDEEEEEFFFFF
===== EEEEEFFFFFGGGGG
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== HHHHHIIIIIJJJJJ
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== IIIIIJJJJJKKKKK
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
===== OOOOOPPPPPQQQQQ
===== PPPPPQQQQQRRRRR
===== QQQQQRRRRRSSSSS
====> _
                                                           X E D I T  1 File
```

Let's say that this isn't the particular line you wanted. To repeat the command, enter:

```
=
```

This directs the editor to repeat the previous command, from this point in the file.

Your screen should now look like this:

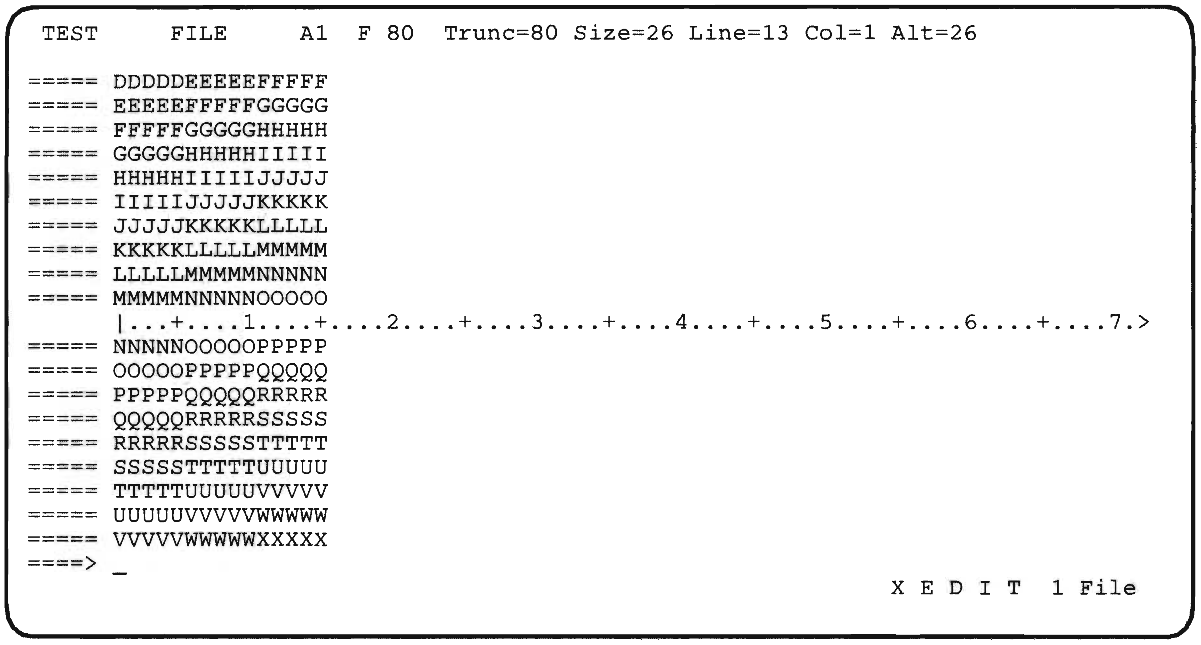


```
 TEST     FILE     A1  F 80  Trunc=80 Size=26 Line=9  Col=1 Alt=26

===== * * * Top of File * * *
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== DDDDDEEEEEFFFFF
===== EEEEEFFFFFGGGGG
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
===== OOOOOPPPPPQQQQQ
===== PPPPPQQQQQRRRRR
===== QQQQQRRRRRSSSSS
===== RRRRRSSSSSTTTTT
====> _
                                                  X E D I T  1 File
```

You can use the **FIND** and **FINDUP** subcommands to locate character strings that occur at the beginning of a line. For example, to find the first occurrence in the file of a line beginning with the string **MMMMM**, enter:

```
f mmmmm
```

Your screen should now look like this:

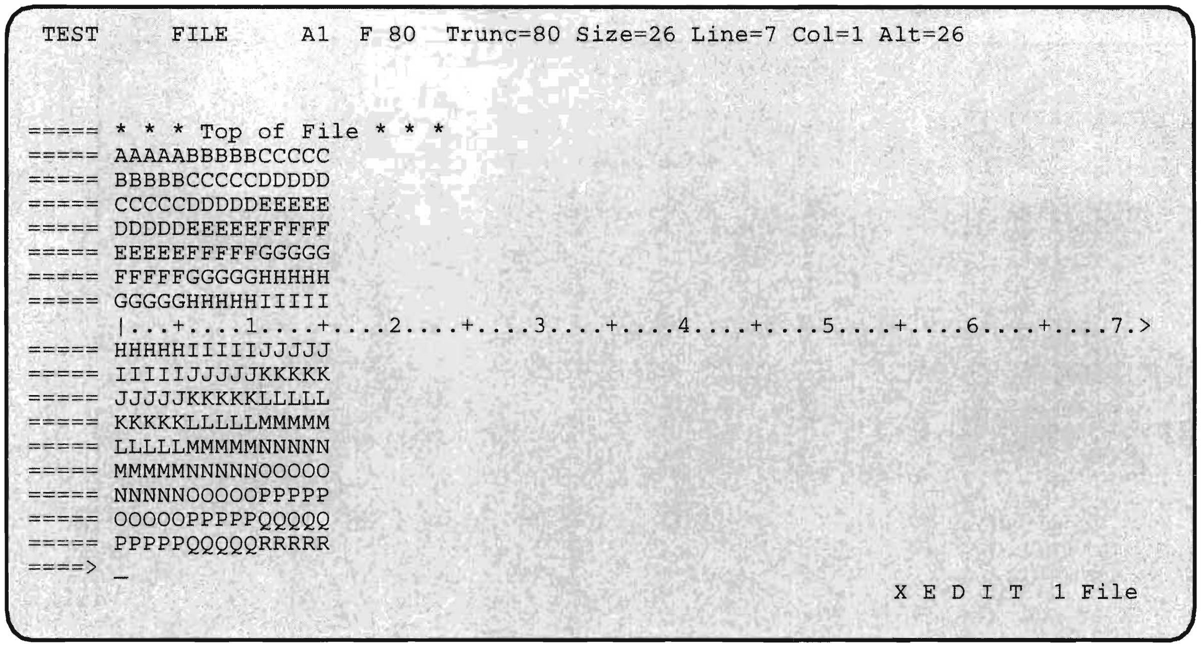


```
 TEST     FILE     A1  F 80  Trunc=80 Size=26 Line=13 Col=1 Alt=26

===== DDDDDEEEEEFFFFF
===== EEEEEFFFFFGGGGG
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== NNNNNOOOOOPPPPP
===== OOOOOPPPPPQQQQQ
===== PPPPPQQQQQRRRRR
===== QQQQQRRRRRSSSSS
===== RRRRRSSSSSTTTTT
===== SSSSSTTTTTUUUUU
===== TTTTTUUUUUVVVVV
===== UUUUUVVVVVWWWWW
===== VVVVVWWWWWXXXXX
====> _
                                                     X E D I T  1 File
```

To find a previous occurrence of a string, use the **findup** subcommand. For example, to find the line beginning with **GGGGG**, enter

```
findu ggggg
```

Your screen should now look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=26 Line=7 Col=1 Alt=26


===== * * * Top of File * * *
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== DDDDDEEEEEFFFFF
===== EEEEEFFFFFGGGGG
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
===== OOOOOPPPPPQQQQQ
===== PPPPPQQQQQRRRRR
====> _
                                                              X E D I T  1 File
```

The **FINDUP** subcommand locates the character string preceding the current line; **FIND** locates the string following the current line.

Using the Prefix Subcommands

Besides the subcommands that can be entered on the command line, there are some that you enter into the prefix area of specific lines you want to manipulate in the file.

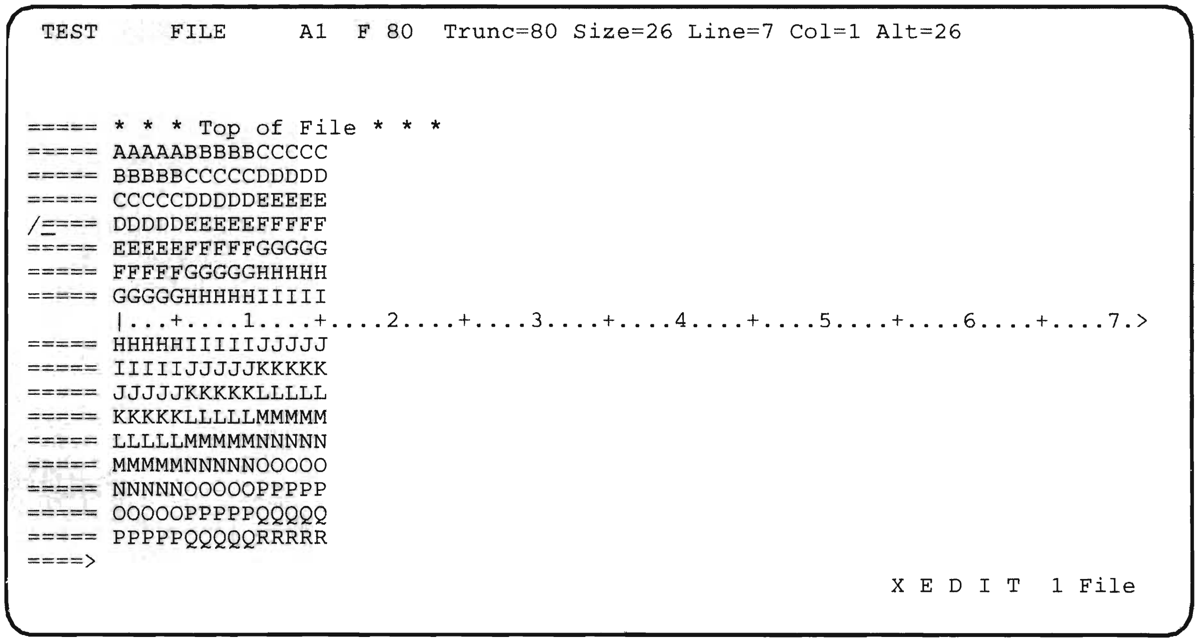
There are three prefix subcommands that can be used to manipulate the display without altering the data. These are:

/	Make this line the current line.
X	Inhibit the display of one or more lines.
S	Restore the display of one or more inhibited lines.

Examples

Let's see how these work. Using the cursor movement keys, position the cursor in the prefix area of the line beginning with the character string DDDDD, and type the slash character. You can enter prefix subcommands anywhere in the prefix area, but for now type the slash (/) in column 1. Don't press ENTER yet, but check your screen with the following example.

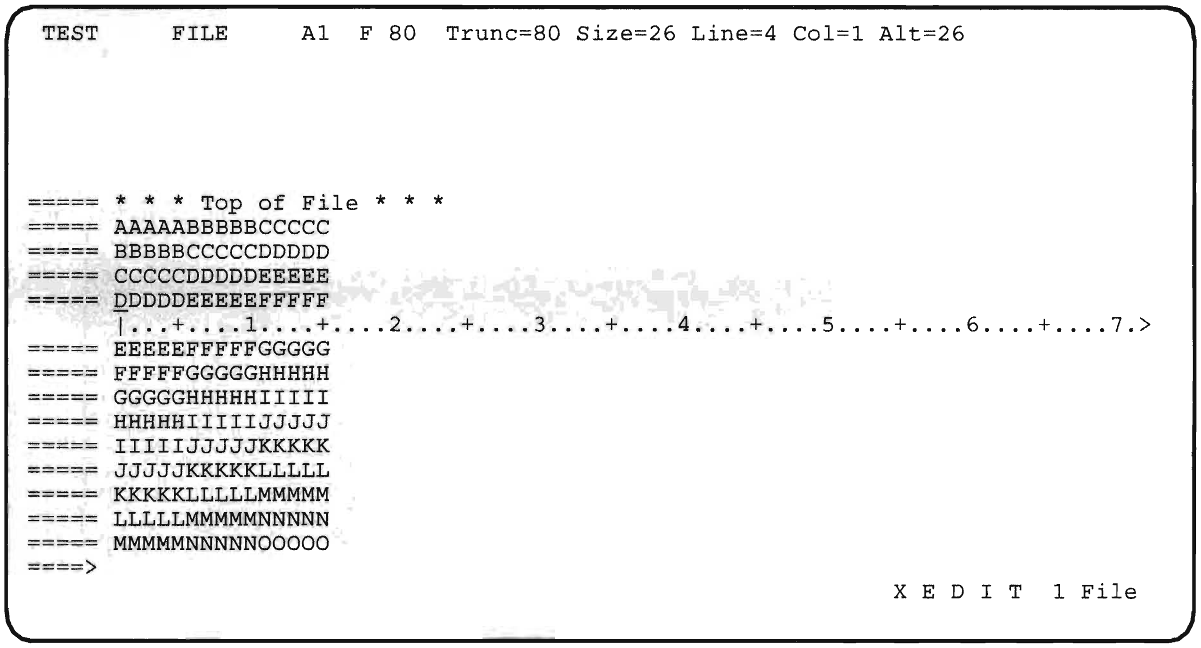
Your screen should now look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=26 Line=7 Col=1 Alt=26


===== * * * Top of File * * *
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
/==== DDDDDEEEEEFFFFF
===== EEEEEFFFFFGGGGG
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
===== OOOOOPPPPPQQQQQ
===== PPPPPQQQQQRRRRR
====>
                                                          X E D I T  1 File
```

Now press ENTER.

Your screen should now look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=26 Line=4 Col=1 Alt=26




===== * * * Top of File * * *
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== DDDDDEEEEEFFFFF
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== EEEEEFFFFFGGGGG
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
====>
                                                        X E D I T  1 File
```

Notice that the cursor is positioned in the first column of the line beginning DDDDD.

The **x** (and **xx**) subcommand is used to **exclude lines** from the display without deleting them from the file. Use the prefix subcommand **S**, to restore the lines to the display. This is useful when you want to compress a file to make it more readable. Also, changes made to a file are not made to excluded lines. This is useful if you want to make global changes in a file, except for specific lines.

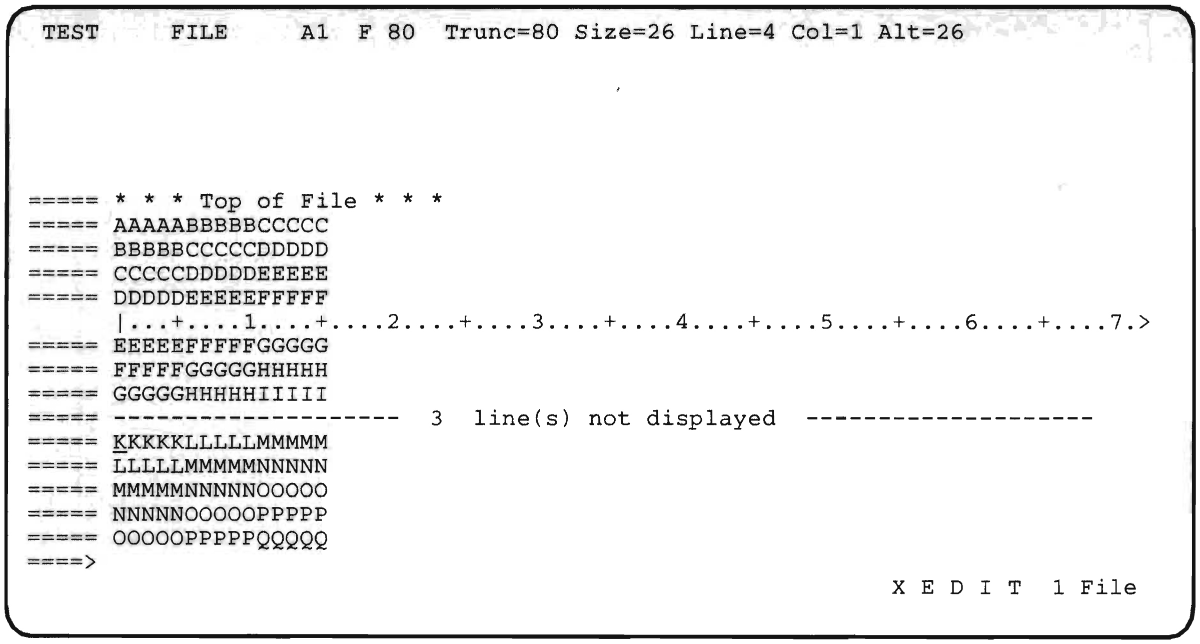
For example, to inhibit the display of the lines containing the character string, JJJJJ move the cursor to the prefix area next to the first occurrence of that line and enter:

```
xx
```

Now move the cursor to the prefix area next to the last line in which the string JJJJJ occurs and enter:

```
xx
```

Your screen should now look like this:



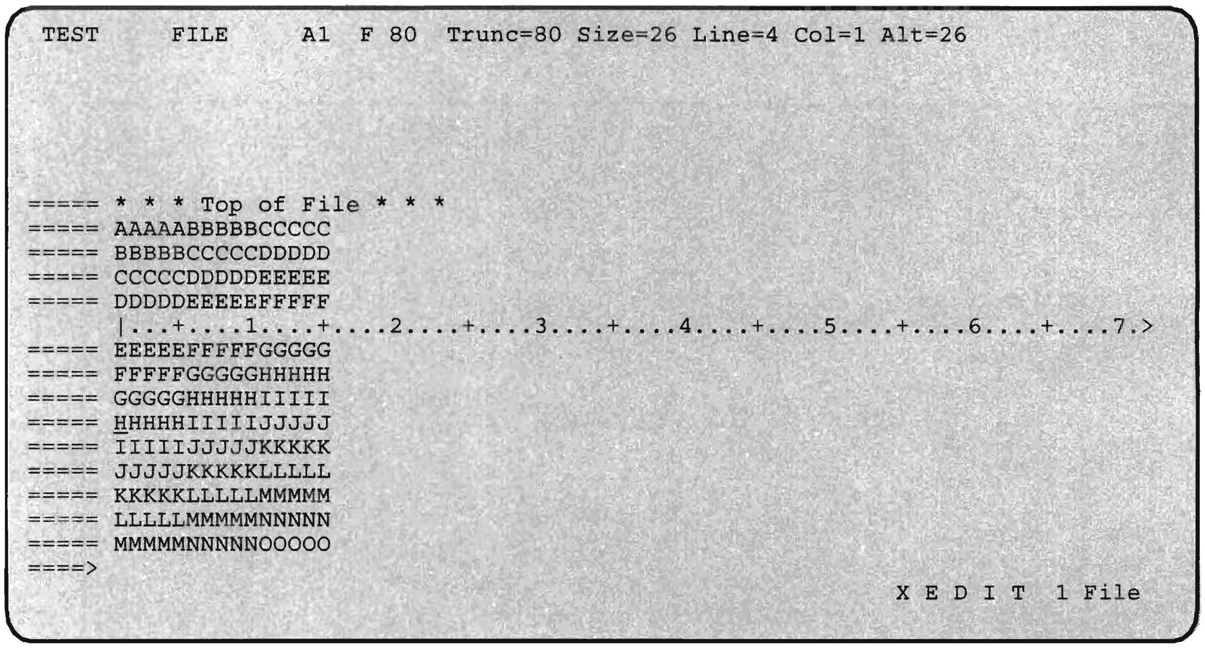
```
 TEST     FILE     A1  F 80  Trunc=80 Size=26 Line=4 Col=1 Alt=26




===== * * * Top of File * * *
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== DDDDDEEEEEFFFFF
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== EEEEEFFFFFGGGGG
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== --------------------  3  line(s) not displayed  --------------------
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
===== OOOOOPPPPPQQQQQ
====>
                                                          X E D I T  1 File
```

A shadow line of dashes and a message appears where the excluded lines were. The message tells you how many lines were excluded.

To restore the display of the excluded lines, enter in the prefix area of the line where the message appears:

```
s
```

Your screen should now look like this:



```
 TEST      FILE      A1  F 80  Trunc=80 Size=26 Line=4 Col=1 Alt=26



===== * * * Top of File * * *
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== DDDDDEEEEEFFFFF
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== EEEEEFFFFFGGGGG
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
====>
                                                           X E D I T  1 File
```

Manipulating Data

The simplest way to change data while editing is to move the cursor to the character that you want to change and type directly over it. For example, suppose you want to change the line beginning HHHHH so that it begins AHAHA. You could position the cursor at the start of the string HHHHH and simply type over it. The line would then read

```
AHAHAIIIIIJJJJJ
```

You can also insert characters into a line between two other characters, or delete characters from the line, by using the **insert** and **delete** keys.

This method of editing a file is simple and direct, and is probably the best way to make simple changes. Some changes you will want to make are less simple, and the direct method of editing may not be appropriate.

In addition to the subcommands which manipulate the display, the editor has a number of subcommands for manipulating data. Some of these can be executed from the prefix area, others from the command line.

Using the Prefix Subcommands

The following list gives the edit subcommands for changing data which can be executed from the prefix area.

A	Add one or more blank lines (equivalent to I).
C	Copy one or more lines to another place in the file.
D	Delete one or more lines from a file.
I	Insert one or more blank lines (equivalent to A).
M	Move one or more lines to another place in the file.
"	Duplicate one or more lines in a file.
>	Shift the data one or more lines n positions to the right.
<	Shift the data one or more lines n positions to the left.

Supplementing the **MOVE** and **COPY** prefix subcommands are two additional subcommands that designate the location where moved or copied lines are to be placed. These are:

F	Designates the line FOLLOWING which the lines are to be placed.
P	Designates the line PRECEDING which the lines are to be placed.

Examples

The following examples are all valid ways for you to enter prefix subcommands:

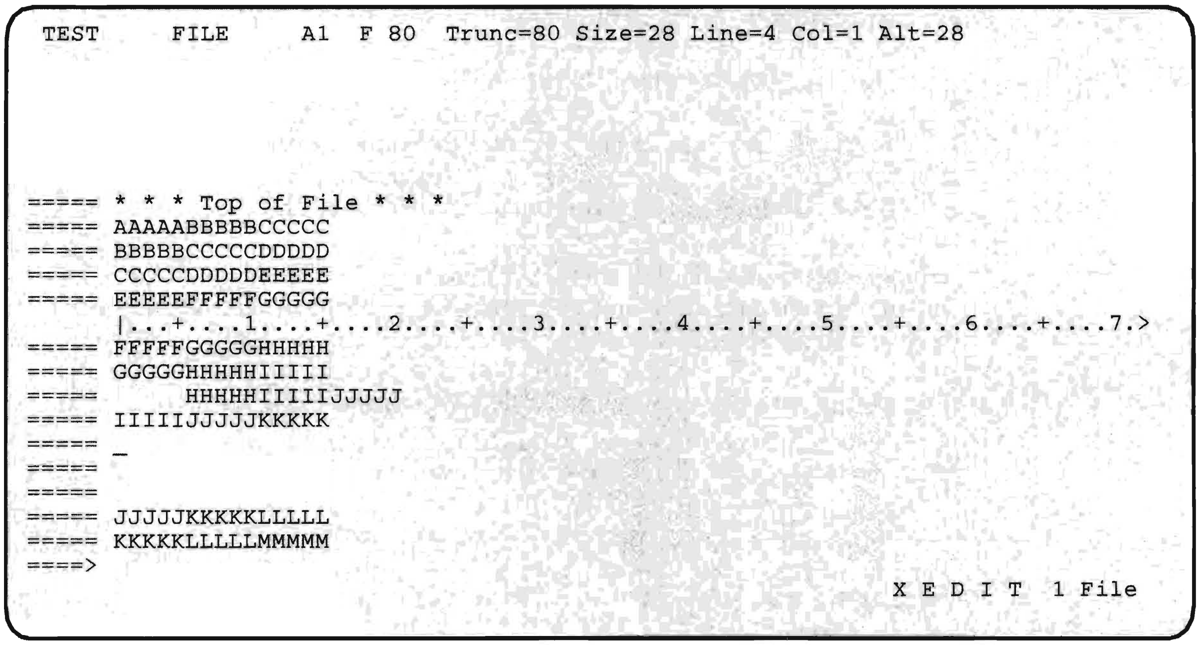
====**A**	Adds one line following the one on which the command is entered.
a3===	Adds three lines
==**D**==	Deletes a line
>====	Shifts the line one space to the right.
==<**12**	Shifts the line twelve spaces to the left.

In the last two examples, these subcommands cause the data itself to be shifted, not just the display. So, data may be lost when you do a shift. Now execute the following prefix subcommands by typing the subcommands indicated below, one after the other. (The full prefix area is shown in each example, together with the line on which the subcommand is to be entered.) Don't press the ENTER key until you've typed all the examples.

- On the line beginning DDDDD, type the prefix subcommand `=d===`
- On the line beginning HHHHH, type the prefix subcommand `==>5=`
- On the line beginning IIIII, type the prefix subcommand `a3===`

Press ENTER.

Your screen should now look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=28 Line=4 Col=1 Alt=28




===== * * * Top of File * * *
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== EEEEEFFFFFGGGGG
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
=====      HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
===== _
=====
=====
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
====>
                                                        X E D I T  1 File
```

Corresponding to each of the examples given, you'll notice the following:

1. The line beginning DDDDD has been deleted (prefix subcommand d).
2. The line beginning HHHHH has been indented 5 spaces to the right (prefix subcommand >).
3. Three blank lines have been added following the line beginning IIIII (prefix subcommand a).

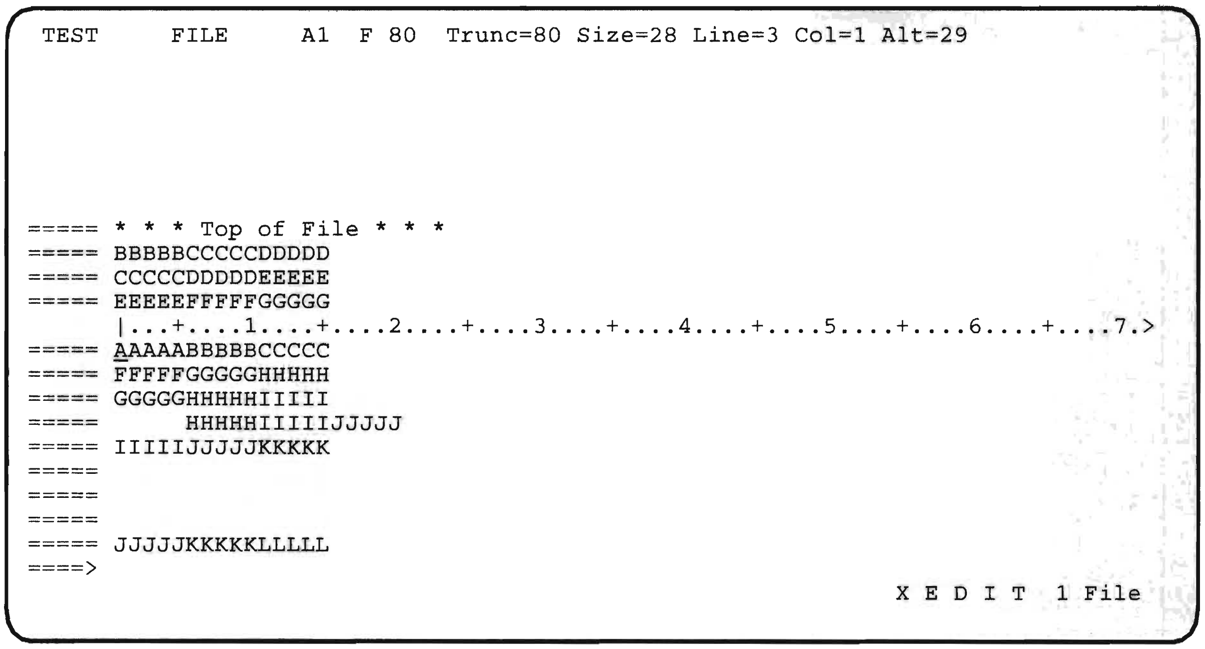
You can use the **C** and **M** prefix subcommands (for **COPY** and **MOVE**) to move or copy lines in the file to another place in the file. For example, type the following prefix subcommand on the line beginning AAAAA (the first line of the file), but don't press ENTER:

```
=m=== AAAAABBBBBCCCCC
```

Now move the cursor to the line beginning EEEEE and type the following prefix subcommand, and press ENTER:

```
f==== EEEEEFFFFFGGGGG
```

Your display should now look like this:



```
  TEST      FILE      A1  F 80  Trunc=80 Size=28 Line=3 Col=1 Alt=29





===== * * * Top of File * * *
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== EEEEEFFFFFGGGGG
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== AAAAABBBBBCCCCC
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
=====      HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
=====
=====
=====
===== JJJJJKKKKKLLLLL
====>
                                                          X E D I T  1 File
```

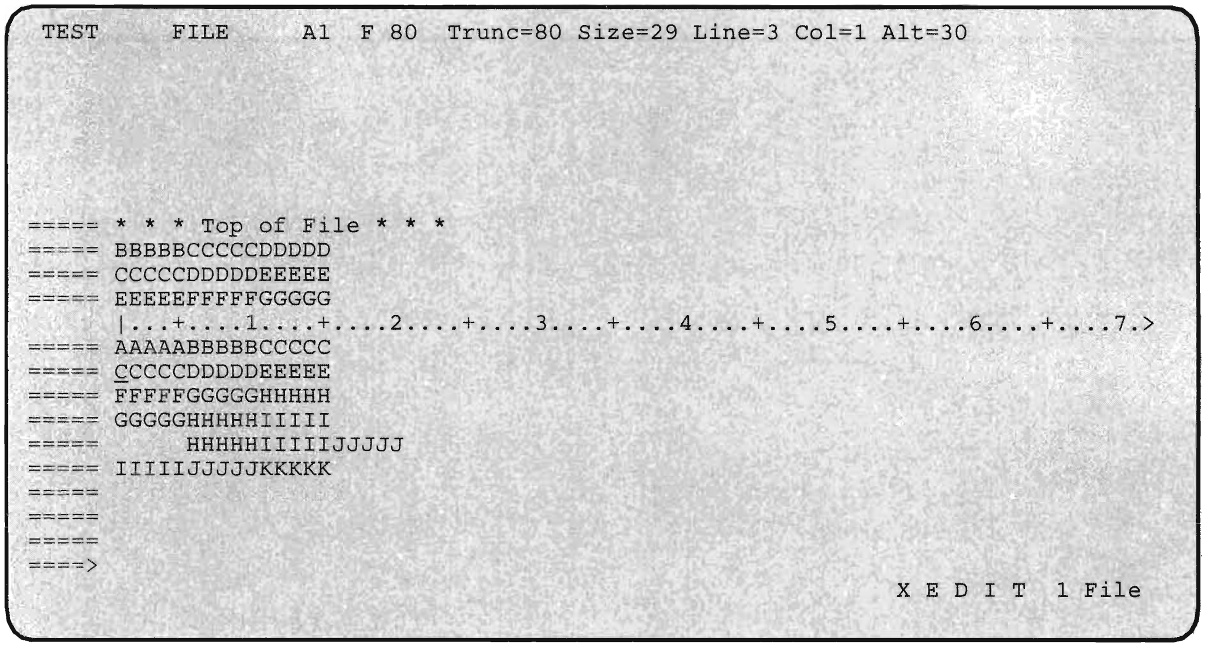
The first line of the file has been moved to follow the fourth line. Now position the cursor on the (new) second line of the file and type the following prefix subcommand (don't press ENTER yet):

```
c==== CCCCDDDDDEEEEE
```

Now move the cursor to the fourth line of the file, type the following prefix subcommand, and press ENTER:

```
==f== AAAAABBBBBCCCCC
```

Your display should now look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=29 Line=3 Col=1 Alt=30







===== * * * Top of File * * *
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== EEEEEFFFFFGGGGG
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== AAAAABBBBBCCCCC
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
=====      HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
=====
=====
=====
====>
                                                            X E D I T  1 File
```

The second line of the file has been copied to follow the fourth line.

Manipulating Blocks of Lines

One of the most powerful aspects of the prefix subcommands is that you can use them to move, copy, duplicate, delete and shift blocks of lines.

If you enter a single **c**, **m**, **"**, **d**, < or > in the prefix area of a specific line, only that line will be affected. If you want to move, copy, duplicate, delete or shift a specific number of lines, you can type the command in the prefix area, followed by a number designating the number of lines to be affected.

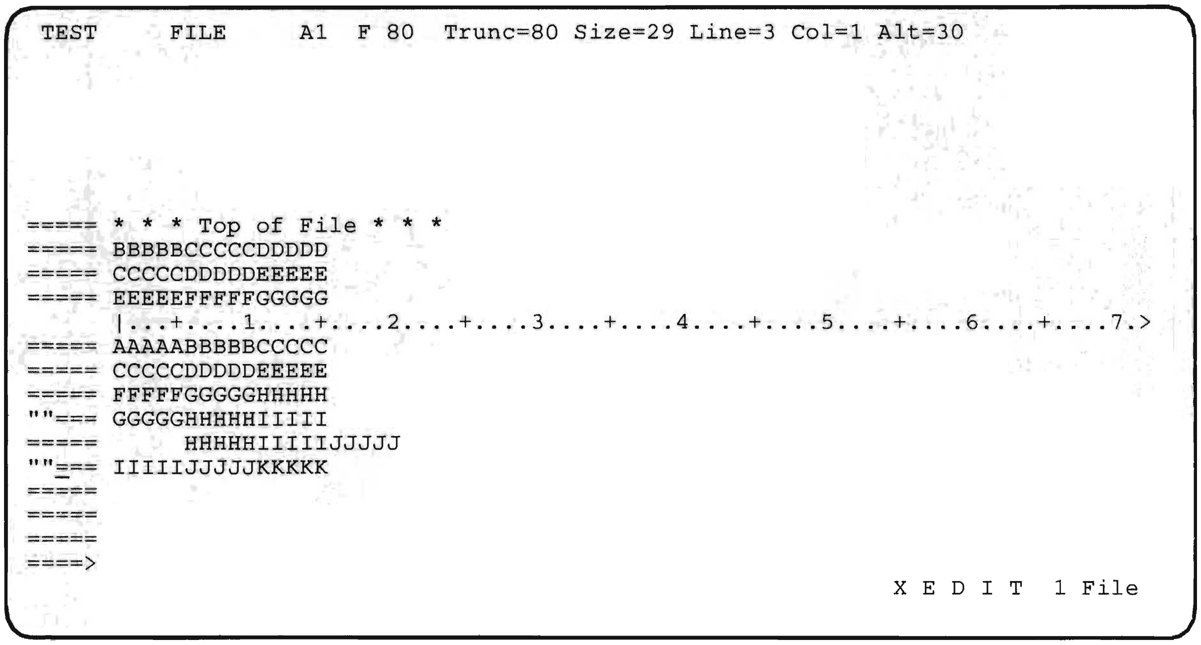
You may find that you want to move a very large block of lines, and that counting the number of lines in the block is tedious or impractical. In the section describing the **x** and **xx** prefix subcommands, you saw that a block of lines could be hidden by placing a double-x on the first line of the block, and another double-x on the last line of the block. The **c**, **m**, **"**, < and > prefix subcommands work the same way.

Examples

For example, to duplicate the block of lines starting with the line beginning GGGGG and ending with the line beginning LLLLL, move the cursor to the line beginning GGGGG and type "" in the prefix area:

```
""=== GGGGGHHHHHIIIII
```

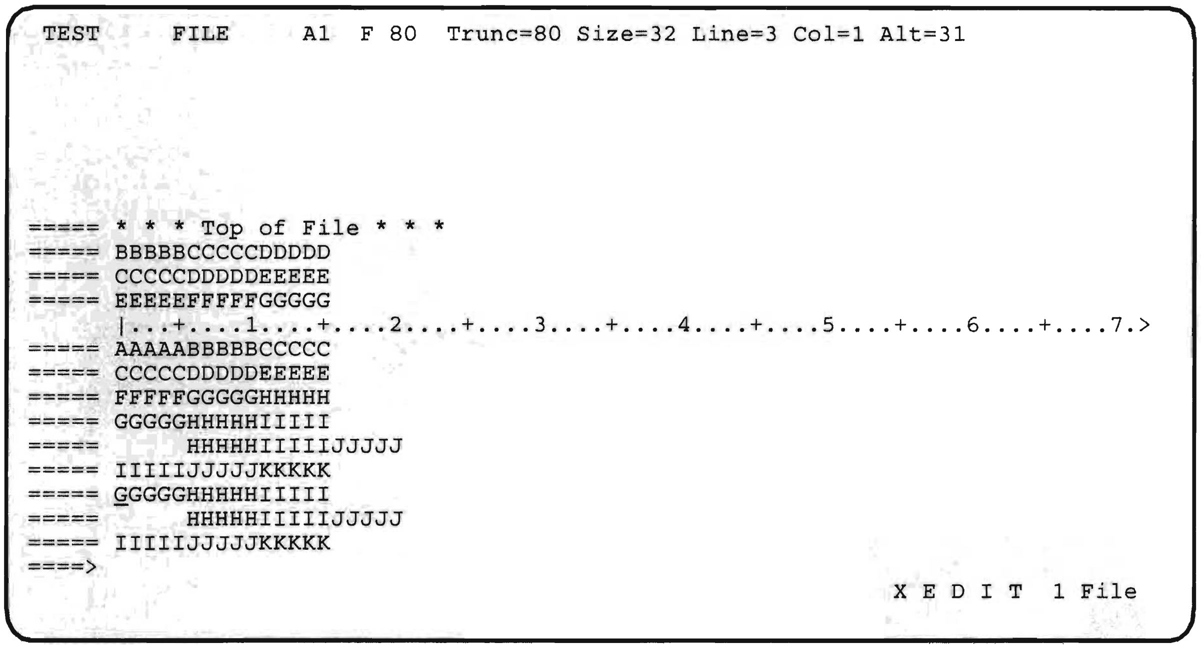
Then move the cursor to the line beginning IIIII and do the same thing; don't press ENTER yet, but check to make sure your screen looks like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=29 Line=3 Col=1 Alt=30




===== * * * Top of File * * *
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== EEEEEFFFFFGGGGG
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== AAAAABBBBBCCCCC
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
""=== GGGGGHHHHHIIIII
=====      HHHHHIIIIIJJJJJ
""=== IIIIIJJJJJKKKKK
=====
=====
=====
====>
                                                      X E D I T  1 File
```

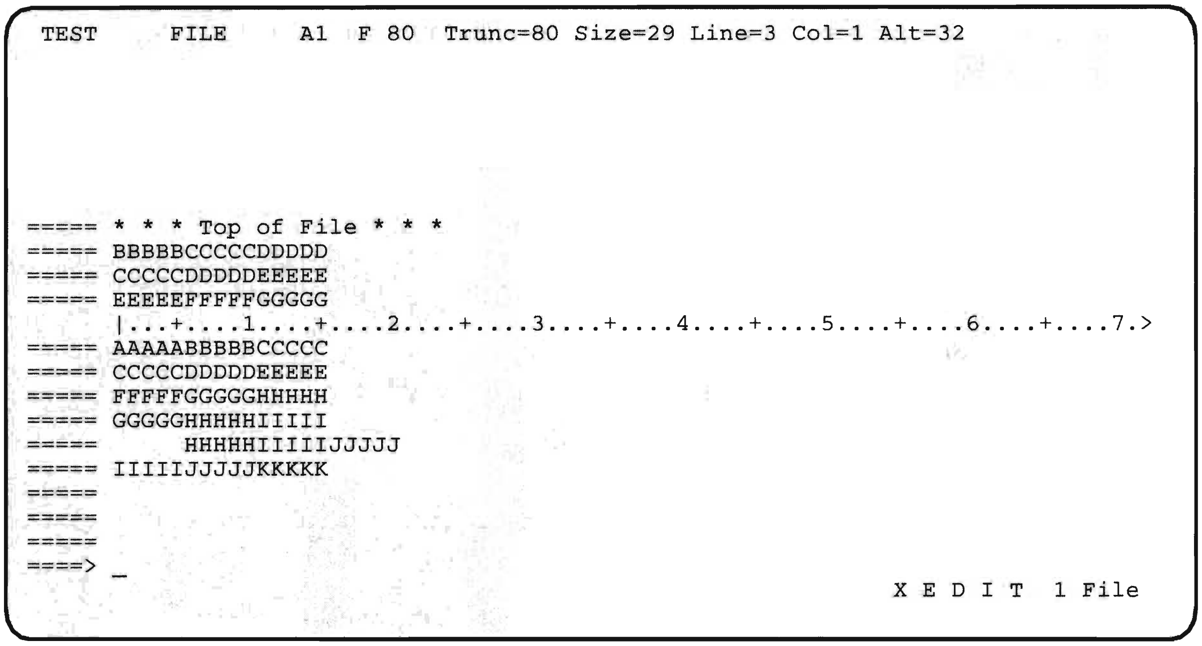
Now press enter.

Your screen should now look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=32 Line=3 Col=1 Alt=31





===== * * * Top of File * * *
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== EEEEEFFFFFGGGGG
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== AAAAABBBBBCCCCC
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
=====      HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
===== GGGGGHHHHHIIIII
=====      HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
====>
                                                          X E D I T  1 File
```

To delete the new block of lines, move the cursor to the first line of the new block and type **dd**. Then move the cursor to the last line of the new block and type **dd**. After pressing ENTER, your screen should look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=29 Line=3 Col=1 Alt=32






===== * * * Top of File * * *
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== EEEEEFFFFFGGGGG
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== AAAAABBBBBCCCCC
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
=====      HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
=====
=====
=====
====> _
                                                          X E D I T  1 File
```

The other block prefix subcommands work the same way.

Using the Command Line Subcommands

Together with the full-screen capabilities of the editor — the ability to make changes to your file directly on the screen image — the prefix subcommands are one of the most powerful tools of the editor.

There are additional subcommands for changing data that can only be entered on the command line. In the list below, some of the subcommands for changing data are given. (Where an abbreviation of a subcommand is permitted, the shortest acceptable version of the name is indicated by upper-case letters.)

Add	Add n lines after the current line.
ALter	Change a single character to another.
Change	Change one string to another.
COpy	Copy one or more lines from one location to another.
DELete	Delete one or more lines.
DUPlicate	Duplicate one or more lines.

GET	Insert lines from another file or from a special buffer (see PUT).
Join	Join two lines.
LOWercase	Change upper-case letters to lower case.
MErge	Combine two sets of lines.
MOve	Move one or more lines to another place in the file.
Overlay	Replace characters in the current line.
PUT	Insert lines into another (new or existing) file.
PUTD	Insert lines into another (new or existing) file and also delete lines from the original file.
RECover	Recover deleted lines.
Replace	Replace the current line with text following; or delete the current line and enter input mode.
SHift	Move data right or left (data loss possible).
SORT	Sort all or part of a file in ascending or descending order.
SPlit	Split a line into two or more lines.
UPPercase	Translates all lower-case characters into upper case.

Some of these are the **LINEMODE** equivalent of the prefix subcommands. For example, the **ADD**, **COPY**, **DELETE**, **DUPLICATE**, **MOVE**, and **SHIFT** subcommands in the list above are all equivalent to the **a**, **c**, **d**, **"**, **m**, < and > prefix subcommands. The other command line subcommands have no prefix equivalents.

All command line subcommands take effect starting with the current line. The editor also has other subcommands that you can use in relation to the position of the cursor on the display screen. For a full description of these, see *VM/SP System Product Editor Command and Macro Reference.*

Using the Change Subcommand

We will now make some changes using the command line subcommands. We'll begin with the **CHANGE** subcommand. The syntax of the **CHANGE** subcommand requires that you specify:

1. The subcommand

2. The string to be changed

3. The string you want it to be changed to.

The default for the **CHANGE** subcommand gives you one change on the current line. You can also specify the number of lines to be affected (including the current line) and the number of occurrences on each line.

Each of the three required elements must be separated by a **delimiter**. The most common delimiter is the slash (/). The slash occurs infrequently in programs and data files and is conveniently placed on the keyboard. You can, however, use any other character as a delimiter, as long as it doesn't appear in either the string to be changed or the replacement string.

As shown in the list above, the abbreviation for CHANGE is C.

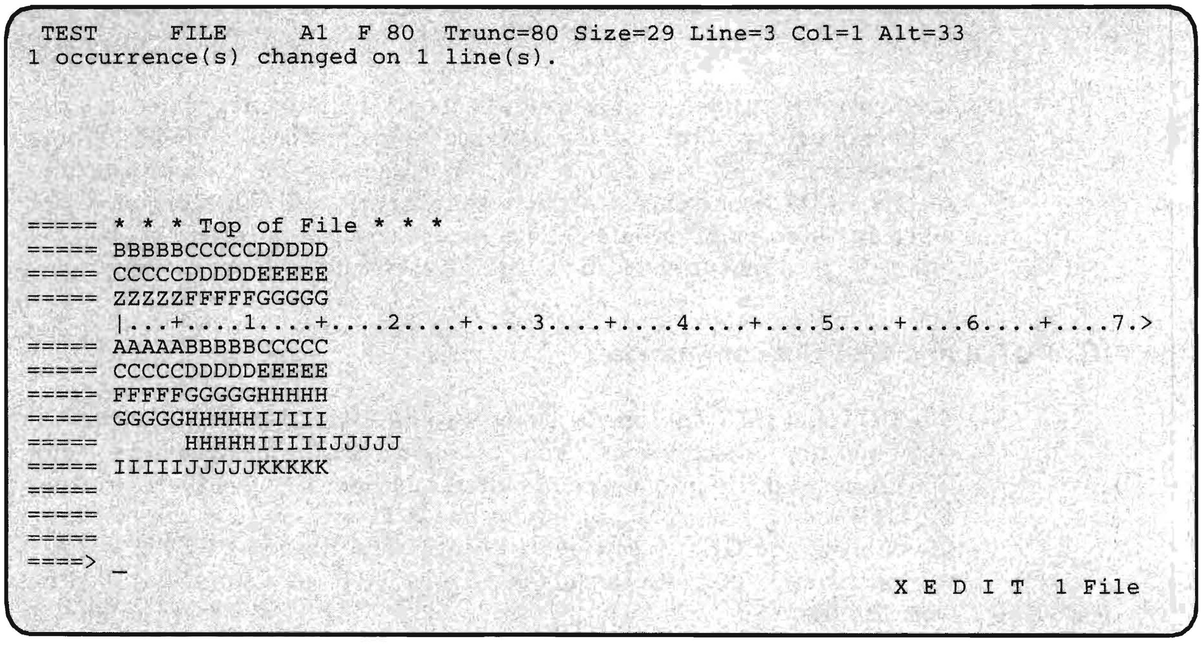
Examples

On the command line enter the following subcommand:

```
c/eeeee/zzzzz/
```

You've changed the string EEEEE to ZZZZZ. Although you entered the subcommand in lower case, the default CASE setting (U) caused the data to be translated into upper case.

Your screen should now look like this:

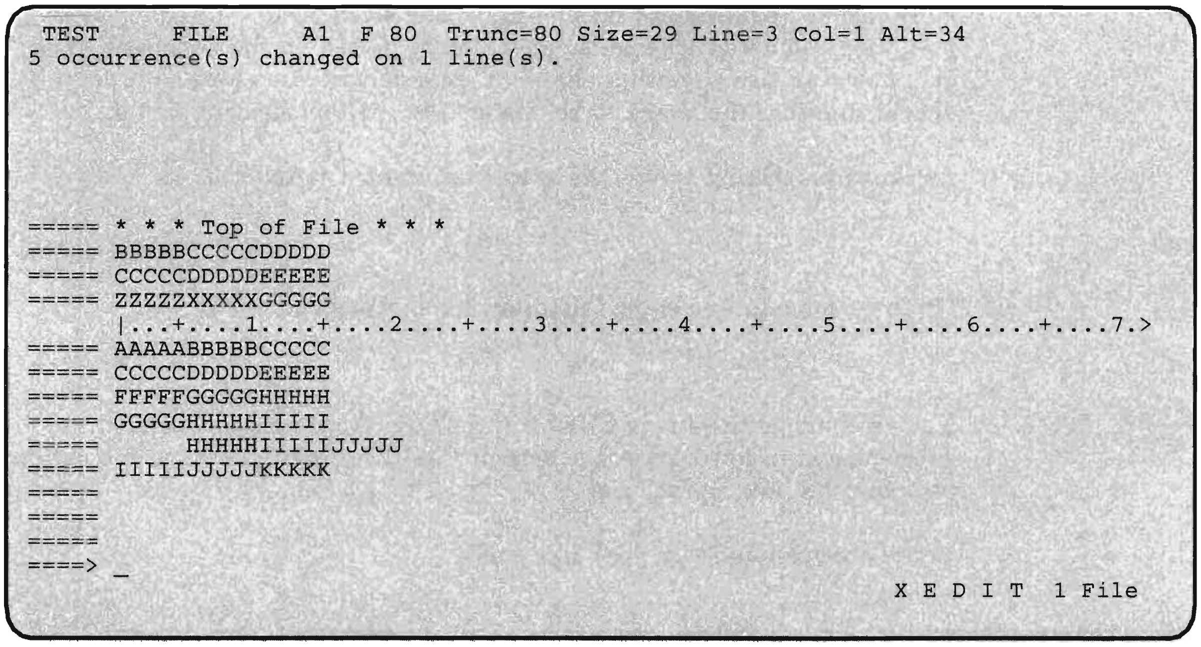


```
 TEST     FILE     A1  F 80  Trunc=80 Size=29 Line=3 Col=1 Alt=33
1 occurrence(s) changed on 1 line(s).




===== * * * Top of File * * *
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== ZZZZZFFFFFGGGGG
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== AAAAABBBBBCCCCC
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
=====      HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
=====
=====
=====
====> _
                                                     X E D I T  1 File
```

To change the string FFFFF on the current line to XXXXX, you can use the form of command:

```
c/f/x/1 5
```

Your screen should now look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=29 Line=3 Col=1 Alt=34
5 occurrence(s) changed on 1 line(s).




===== * * * Top of File * * *
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== ZZZZZXXXXXGGGGG
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== AAAAABBBBBCCCCC
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
=====      HHHHHIIIIIJJJJJ
===== IIIIIJJJJJKKKKK
=====
=====
=====
====> _
                                                          X E D I T  1 File
```

If you want to make a change that affects all the remaining lines in a file (a global change), you can use the asterisk (*) instead of the number-of-lines parameter. The asterisk means "all". You can also use the asterisk to specify all the occurrences on the lines to be affected. The command **c/string1/string2/* *** would change all occurrences of "string1" into "string2" throughout each line of the file, starting with the current line.

Using the PUT, PUTD, and GET Subcommands

The **PUT** and **GET** subcommands are two of the most powerful of the command line subcommands. You can use the **PUT** subcommand to copy lines from your file into a special buffer, and later retrieve them by using the **GET** subcommand. You can also use **PUT** to copy lines directly into a file on disk, and **GET** to retrieve all or part of a file. The **PUTD** subcommand performs the same function as **PUT** but also deletes the lines from the file.

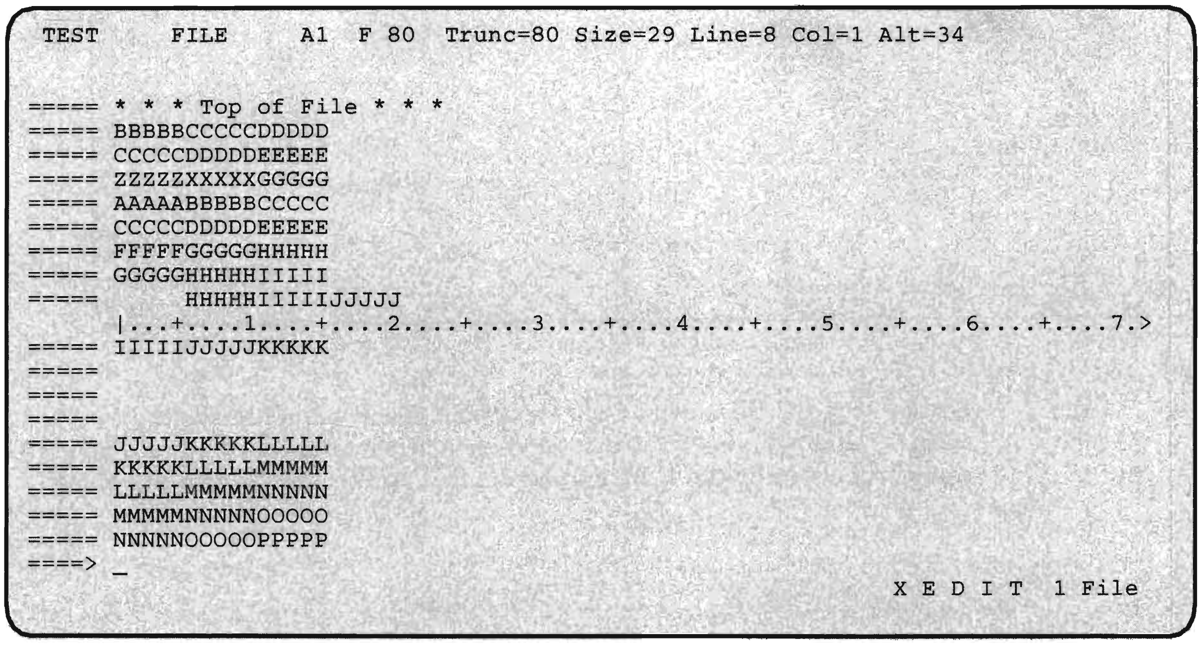
Examples

We'll now **PUT** five lines into a special buffer. Enter:

```
put 5
```

You've now copied five lines into a temporary buffer in the editor. The lines are also present in the file. You will notice that the current line pointer has moved to the line following the block.

Your screen should now look like this:



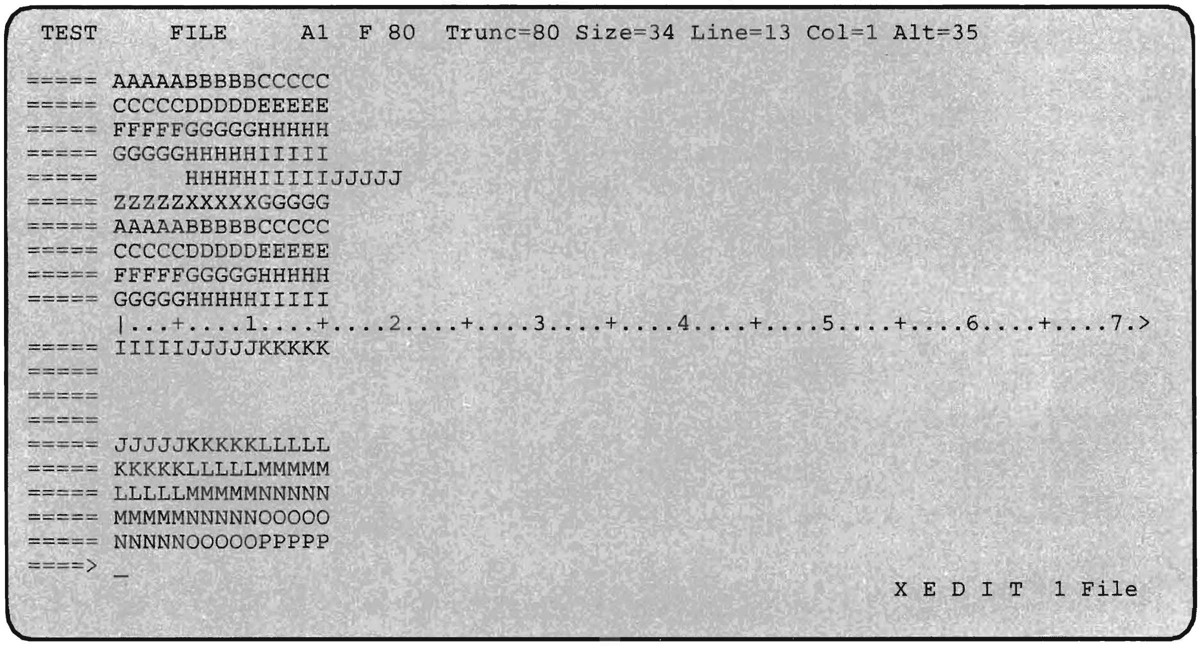
```
 TEST     FILE     A1  F 80  Trunc=80 Size=29 Line=8 Col=1 Alt=34


===== * * * Top of File * * *
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== ZZZZZXXXXXGGGGG
===== AAAAABBBBBCCCCC
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
=====      HHHHHIIIIIJJJJJ
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== IIIIIJJJJJKKKKK
=====
=====
=====
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
====> _
                                                        X E D I T  1 File
```

Now enter the following subcommand:

```
get
```

The five lines you **PUT** have been retrieved from the special buffer and have been inserted into your file following the current line.

Your screen should now look like this:



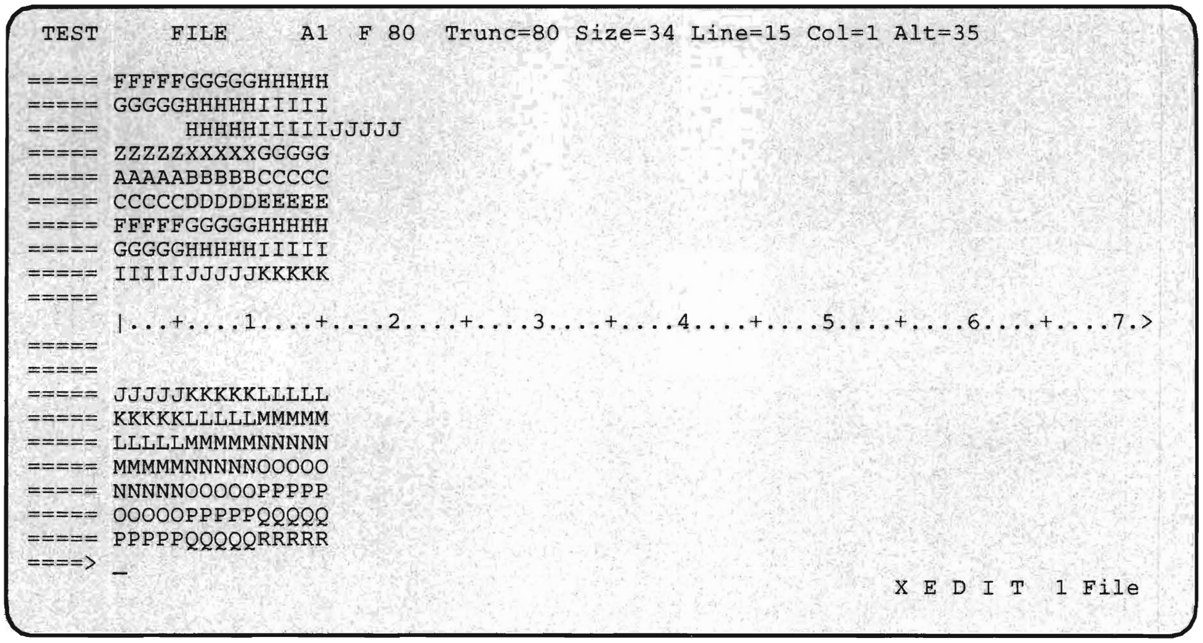
```
 TEST     FILE     A1  F 80  Trunc=80 Size=34 Line=13 Col=1 Alt=35

===== AAAAABBBBBCCCCC
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
=====      HHHHHIIIIIJJJJJ
===== ZZZZZXXXXXGGGGG
===== AAAAABBBBBCCCCC
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== IIIIIJJJJJKKKKK
=====
=====
=====
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
====> _
                                                          X E D I T  1 File
```

By the way, the lines copied via the **PUT** subcommand are still in the special buffer. If you wanted to retrieve a second copy of the lines, you would simply issue the **GET** subcommand again at the appropriate place.

If you want to delete the block of lines as well as make a copy of them in the special buffer or a file on disk, use the **PUTD** subcommand. Let's delete two blank lines from the file and write them to a file on the A-disk. First, issue the following command to position the current line at the first of the blank lines.

```
down 2
```

Your screen should now look like this:

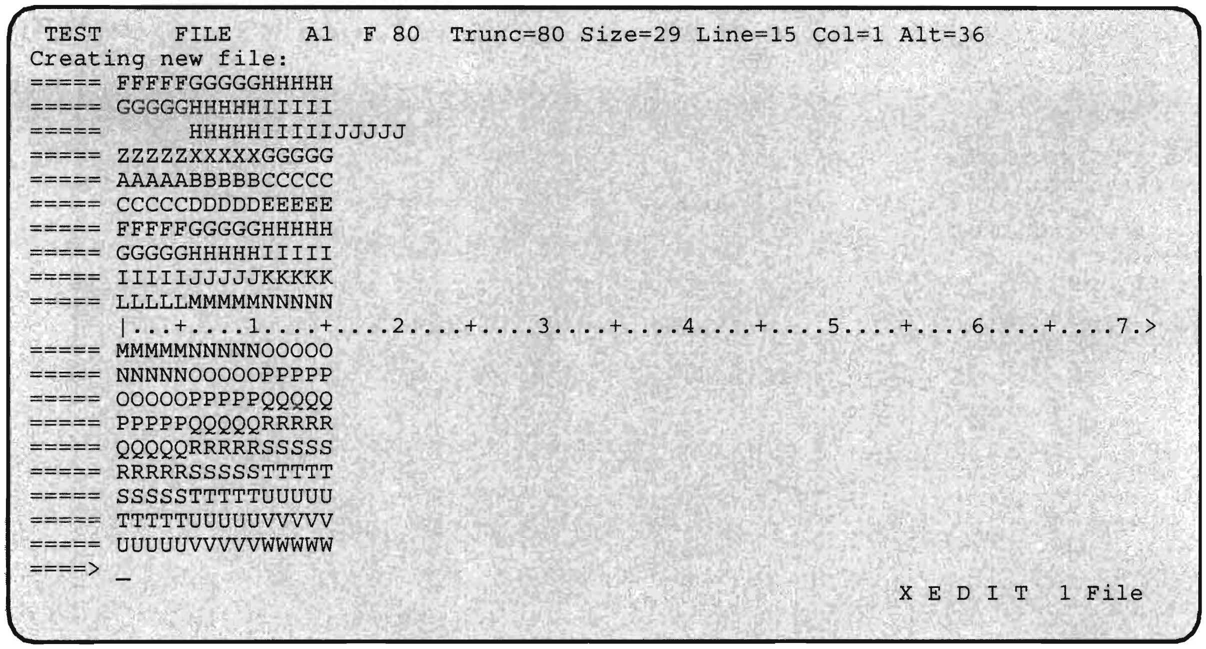


```
 TEST     FILE     A1  F 80  Trunc=80 Size=34 Line=15 Col=1 Alt=35

===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
=====      HHHHHIIIIIJJJJJ
===== ZZZZZXXXXXGGGGG
===== AAAAABBBBBCCCCC
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== IIIIIJJJJJKKKKK
=====
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
=====
=====
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
===== OOOOOPPPPPQQQQQ
===== PPPPPQQQQQRRRRR
====> _
                                                          X E D I T  1 File
```

In the next example, we'll make a deliberate "mistake" in order to demonstrate another aspect of the **GET** subcommand. Instead of deleting only the two blank lines, we'll delete five lines, and then GET back three of them. Now issue the **PUTD** command to copy five lines from the file being edited to a new file called 'TEMP FILE A':

```
putd 5 temp file a
```

Your screen should now look like this:



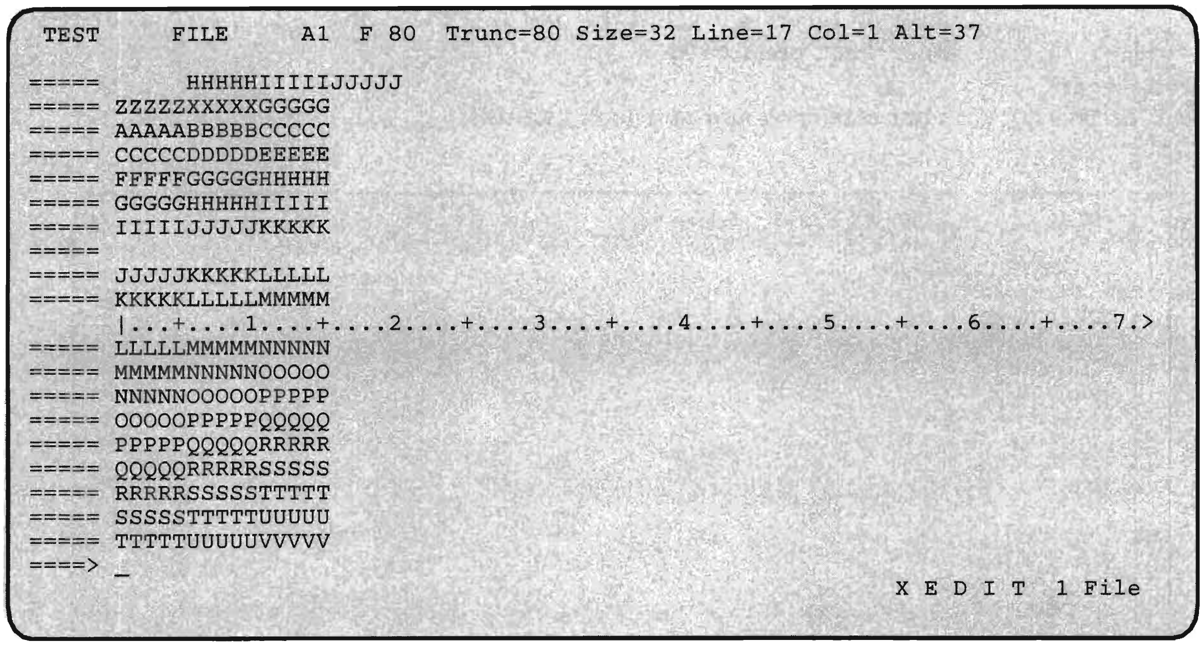
```
 TEST     FILE     A1  F 80  Trunc=80 Size=29 Line=15 Col=1 Alt=36
Creating new file:
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
=====      HHHHHIIIIIJJJJJ
===== ZZZZZXXXXXGGGGG
===== AAAAABBBBBCCCCC
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== IIIIIJJJJJKKKKK
===== LLLLLMMMMMNNNNN
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
===== OOOOOPPPPPQQQQQ
===== PPPPPQQQQQRRRRR
===== QQQQQRRRRRSSSSS
===== RRRRRSSSSSTTTTT
===== SSSSSTTTTTUUUUU
===== TTTTTUUUUUVVVVV
===== UUUUUVVVVVWWWWW
====> _
                                                      X E D I T  1 File
```

We have deleted the two blank lines, but also three lines following it. To recover the three lines from the file TEMP FILE A, we could issue the subcommand GET TEMP FILE A, which would retrieve the entire file, and then delete the two blank lines, using the **d** prefix subcommand or the **delete** subcommand from the command line. Instead, we'll GET only the three lines deleted by mistake. First, we'll position the current line so that the lines we'll be GETting will occur in the right place in the file. Then, we'll issue a variation of the **GET** subcommand. Issue the following subcommands from the command line:

```
up 1
get temp file a 3 3
```

in order to retrieve three of the five lines (that is, lines 3 through 5 of TEMP FILE A). The first number following the filemode is the starting number of the line in the file you specified, relative to the beginning of the file. The second number specifies the number of lines you wish to GET.

Your screen should now look like this:



```
 TEST      FILE      A1  F 80  Trunc=80 Size=32 Line=17 Col=1 Alt=37

=====      HHHHHIIIIIJJJJJ
===== ZZZZZXXXXXGGGGG
===== AAAAABBBBBCCCCC
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== IIIIIJJJJJKKKKK
=====
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
===== OOOOOPPPPPQQQQQ
===== PPPPPQQQQQRRRRR
===== QQQQQRRRRRSSSSS
===== RRRRRSSSSSTTTTT
===== SSSSSTTTTTUUUUU
===== TTTTTUUUUUVVVVV
====> _
                                                          X E D I T  1 File
```

By using similar techniques, you can build programs, documents, and data files from files containing standard routines and paragraphs that you expect to use more than once.

Using Split and Join

Three useful commands, especially when you're writing documentation, are **SPLIT**, **JOIN**, and the combined version **SPLTJOIN**.

These subcommands let you divide a line at a given point or join the next line to the current line at a given point. We'll now **SPLIT** the current line, which is the last line of the block you just retrieved with the **GET** subcommand. Suppose you want to split the line at the string LLLLL, making the current line read KKKKK and the next line LLLLLMMMMM. One way you could do this would be to enter the **SPLIT** subcommand on the command line, specifying the string at which the split is to take place. If we were going to use a subcommand to do this, it would look like this.

Note: Don't enter this.

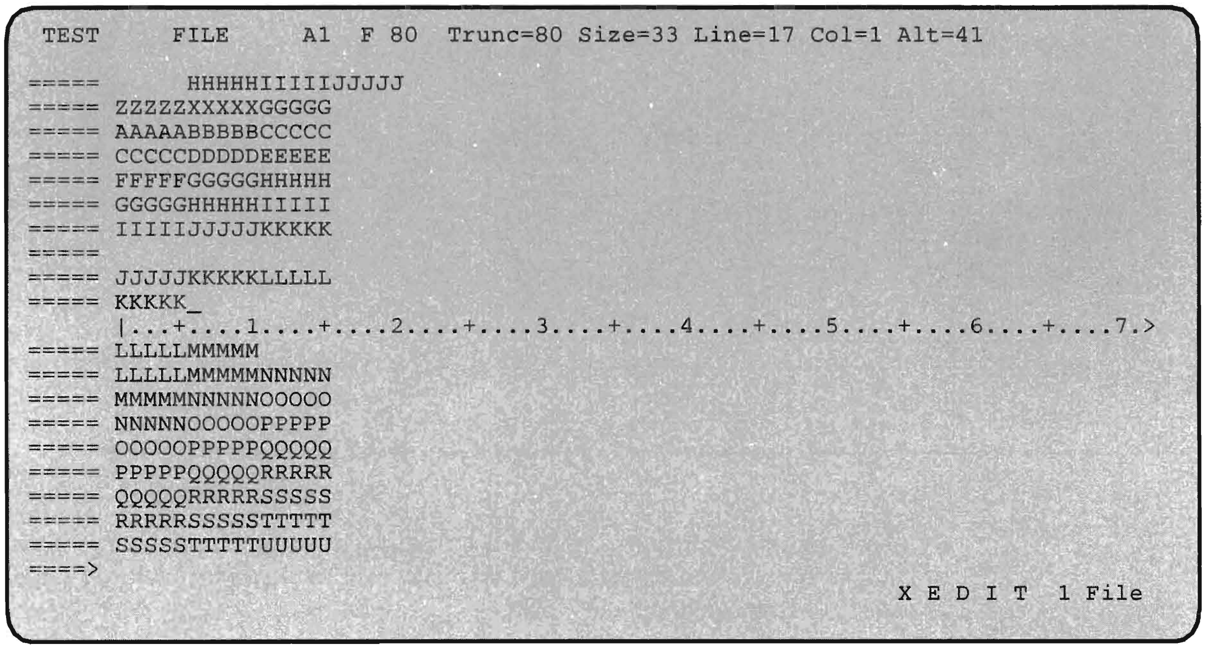
```
split/lllll/
```

The **SPLTJOIN** subcommand lets you split a line, and then rejoin it again if you wish. Your **PF11** key has been preset by the editor to execute the **SPLTJOIN** subcommand.

Examples

To use this form of the subcommand, use the cursor movement keys to position the cursor below the first L of the string LLLLL on the current line. Then press **PF11**.

Your screen should now look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=33 Line=17 Col=1 Alt=41

=====      HHHHHIIIIIJJJJJ
===== ZZZZZXXXXXGGGGG
===== AAAAABBBBBCCCCC
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== IIIIIJJJJJKKKKK
=====
===== JJJJJKKKKKLLLLL
===== KKKKK_
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== LLLLLMMMMM
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
===== OOOOOPPPPPQQQQQ
===== PPPPPQQQQQRRRRR
===== QQQQQRRRRRSSSSS
===== RRRRRSSSSSTTTTT
===== SSSSSTTTTTUUUUU
====>
                                                          X E D I T  1 File
```

Notice that the cursor remains in the same place it was before you pressed the **PF11** key.

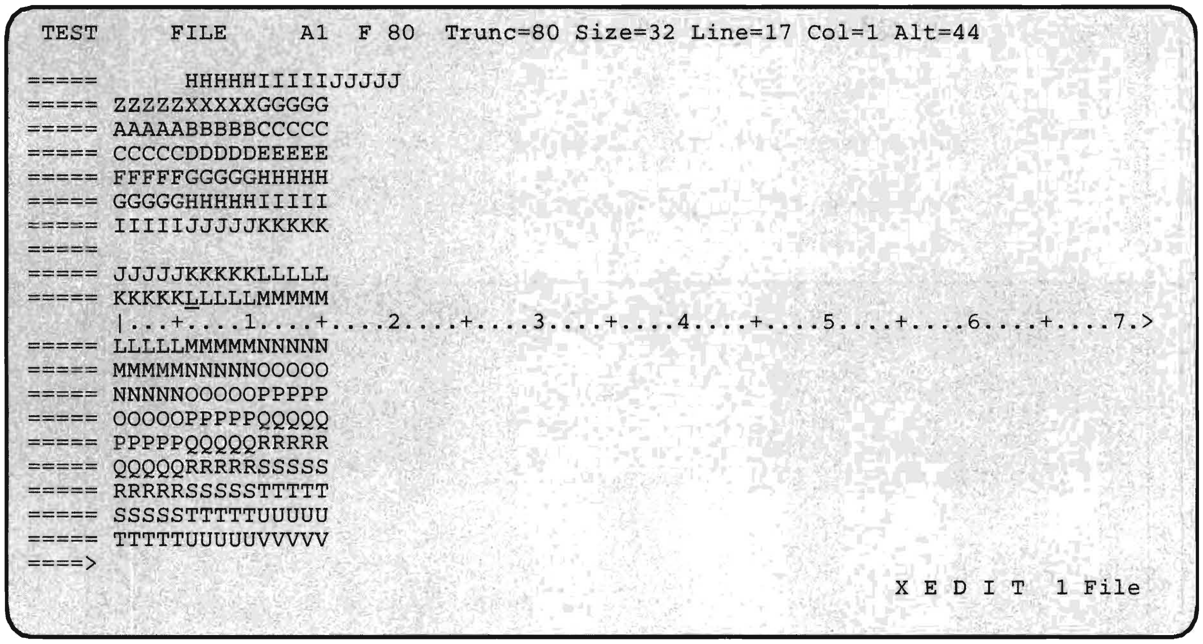
The **JOIN** command joins the current line with the line following. To **JOIN** the split line up again you could use the command line subcommand.

Note: Don't enter this.

```
join
```

Instead, you can simply press **PF11** a second time. Do so now.

Your screen should now look like this:



```
 TEST      FILE      A1  F 80  Trunc=80 Size=32 Line=17 Col=1 Alt=44

=====      HHHHHIIIIIJJJJJ
===== ZZZZZXXXXXGGGGG
===== AAAAABBBBBCCCCC
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== IIIIIJJJJJKKKKK
=====
===== JJJJJKKKKKLLLLL
===== KKKKKLLLLLMMMMM
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== LLLLLMMMMMNNNNN
===== MMMMMNNNNNOOOOO
===== NNNNNOOOOOPPPPP
===== OOOOOPPPPPQQQQQ
===== PPPPPQQQQQRRRRR
===== QQQQQRRRRRSSSSS
===== RRRRRSSSSSTTTTT
===== SSSSSTTTTTUUUUU
===== TTTTTUUUUUVVVVV
====>
                                                          X E D I T  1 File
```

Using the Sort Subcommand

Before we go on to discuss the next set of subcommands, there's one powerful subcommand that is especially useful in working with data files. This is **SORT**, which can be used to reorder all or some of the records in a file.

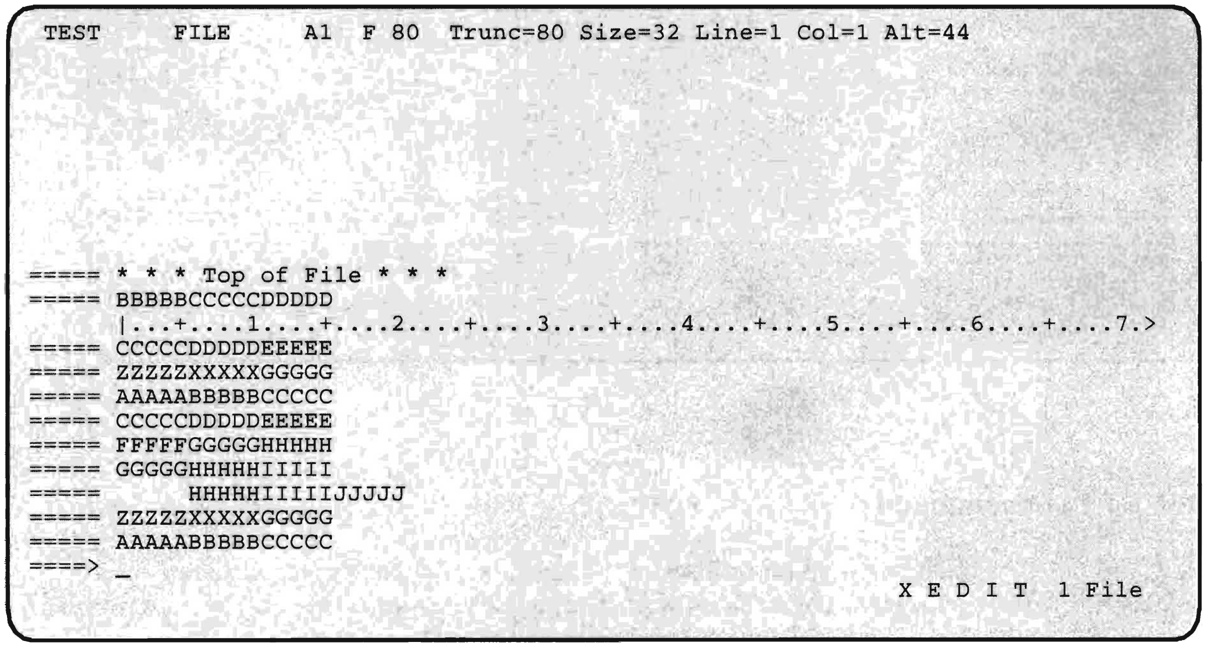
Example

We'll sort only the first ten records in the file. To do this, press ENTER and enter the subcommand:

```
:1
```

to reposition the current line at the first line of the file.

Your screen should now look like this:



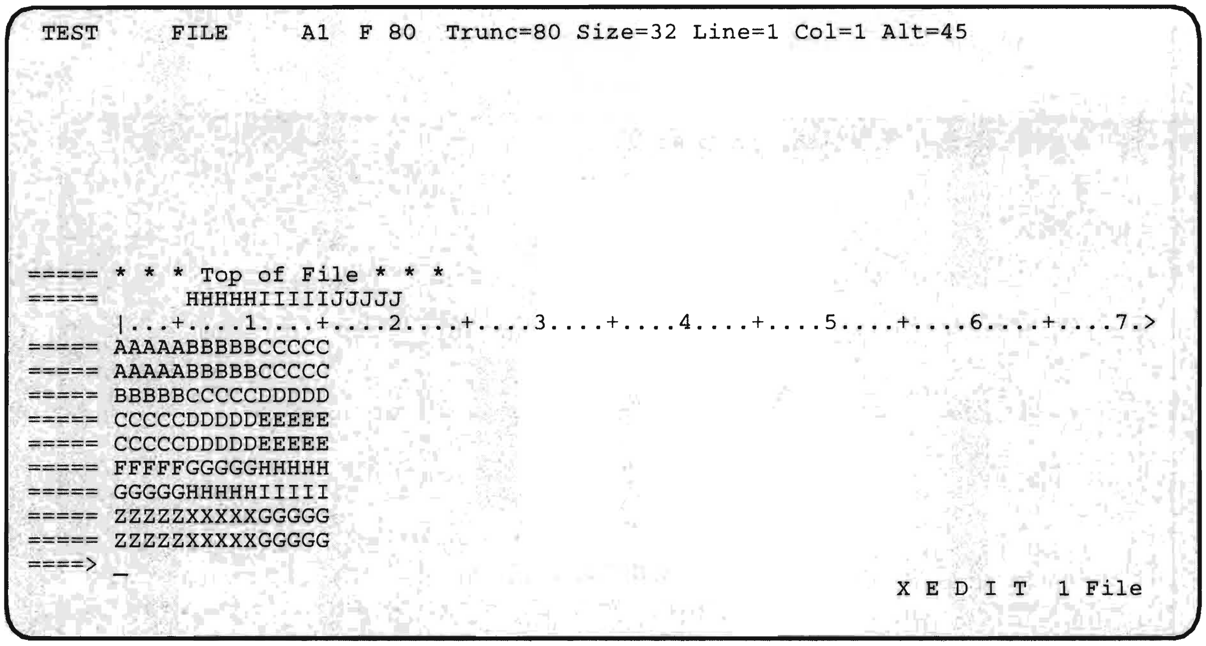
```
 TEST     FILE     A1  F 80  Trunc=80 Size=32 Line=1 Col=1 Alt=44






===== * * * Top of File * * *
===== BBBBBCCCCCDDDDD
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== CCCCCDDDDDEEEEE
===== ZZZZZXXXXXGGGGG
===== AAAAABBBBBCCCCC
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
=====      HHHHHIIIIIJJJJJ
===== ZZZZZXXXXXGGGGG
===== AAAAABBBBBCCCCC
====> _
                                                           X E D I T  1 File
```

Now enter:

```
sort 10 1 5
```

This sorts the first 10 lines of the file, using columns 1 through 5 as the sort key.

Your screen should now look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=32 Line=1 Col=1 Alt=45




===== * * * Top of File * * *
=====      HHHHHIIIIIJJJJJ
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== AAAAABBBBBCCCCC
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== ZZZZZXXXXXGGGGG
===== ZZZZZXXXXXGGGGG
====> _
                                                     X E D I T  1 File
```

If you had wanted to sort the entire file, you could have used an asterisk ('*') instead of the 10. If you had wanted to sort on more than one key, you could have specified additional column pairs.

Editing Multiple Files

Using the editor, you can edit more than one file at a time. This can be very useful, for instance, if you want to examine a compiler listing and compare it with the source code, or if you want to compare two versions of a source program.

We'll do some editing of multiple files. Enter the subcommand:

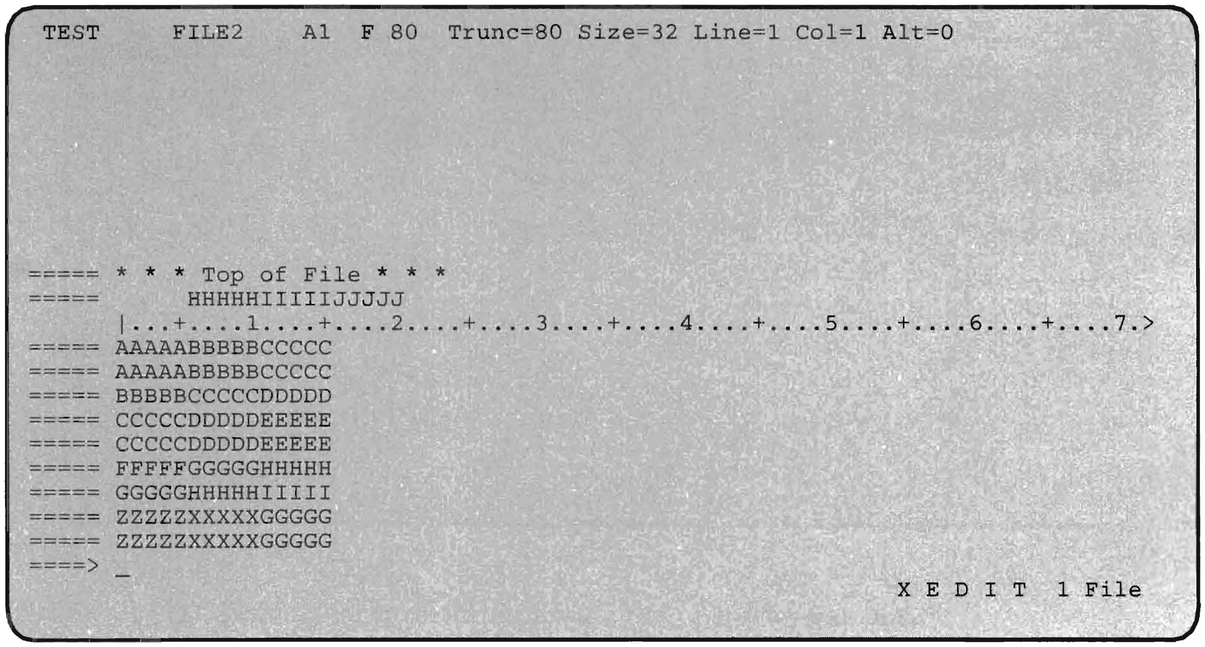
```
save
```

This writes the file TEST FILE A to the A-disk, at the same time keeping a copy in storage for the editor. Now enter the subcommand:

```
ft file2
```

The **FT** subcommand changed the filetype of the file in storage to FILE2.

Your screen should now look like this:



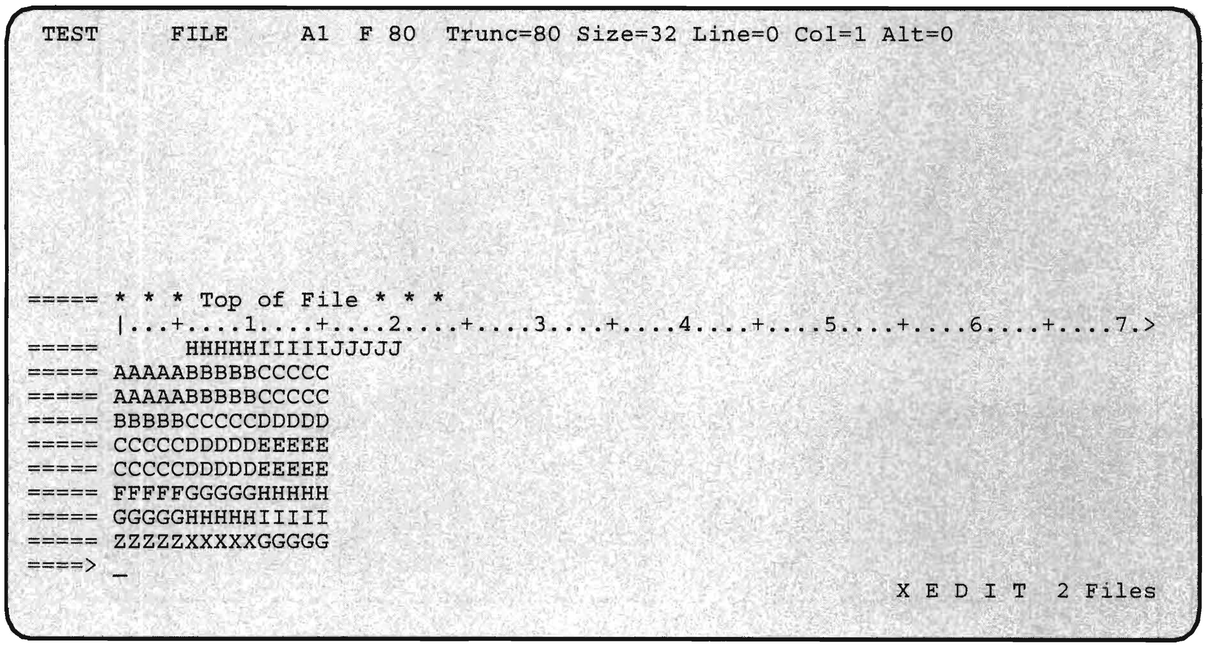
```
 TEST     FILE2    A1  F 80  Trunc=80 Size=32 Line=1 Col=1 Alt=0








===== * * * Top of File * * *
=====      HHHHHIIIIIJJJJJ
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== AAAAABBBBBCCCCC
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== ZZZZZXXXXXGGGGG
===== ZZZZZXXXXXGGGGG
====> _
                                                          X E D I T  1 File
```

The original file (TEST FILE A) has been written to disk.

Now let's edit both TEST FILE A and TEST FILE2 A. To bring TEST FILE A into the edit session, enter the following subcommand on the command line:

```
x test file a (noprof
```

Your screen should now look like this:

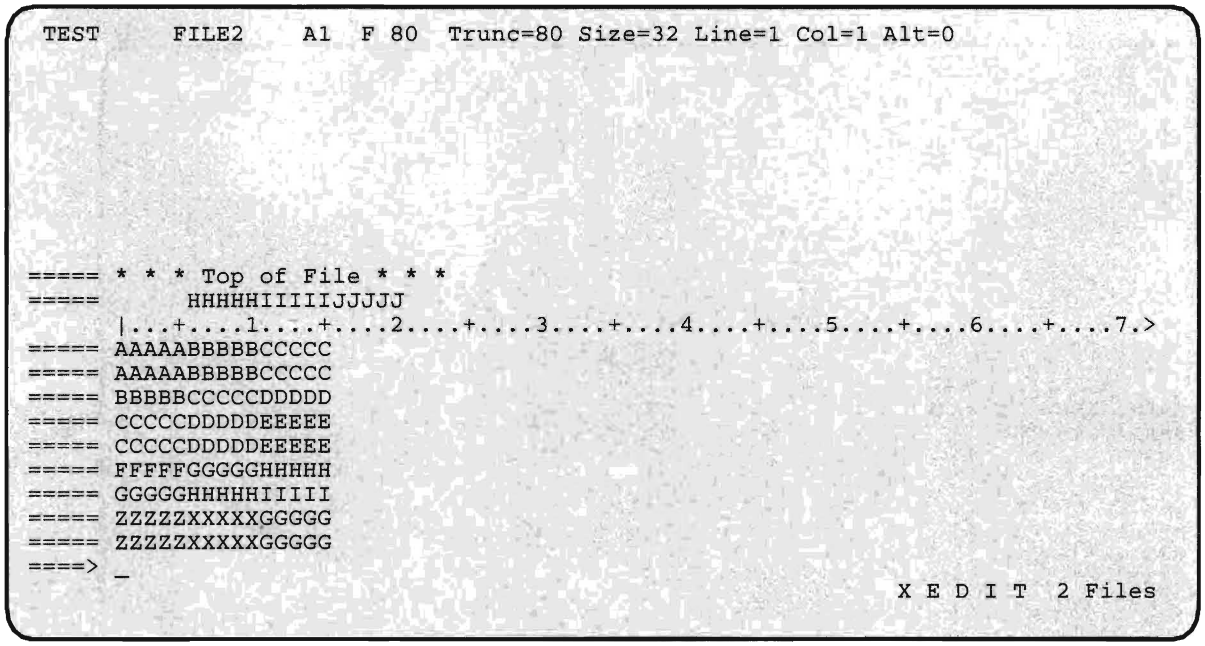


```
 TEST     FILE     A1  F 80  Trunc=80 Size=32 Line=0 Col=1 Alt=0







===== * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
=====      HHHHHIIIIIJJJJJ
===== AAAAABBBBBCCCCC
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== ZZZZZXXXXXGGGGG
====> _
                                                         X E D I T  2 Files
```

To verify that you still have TEST FILE2 A in the editor, enter the subcommand:

```
x
```

Your screen should now look like this:



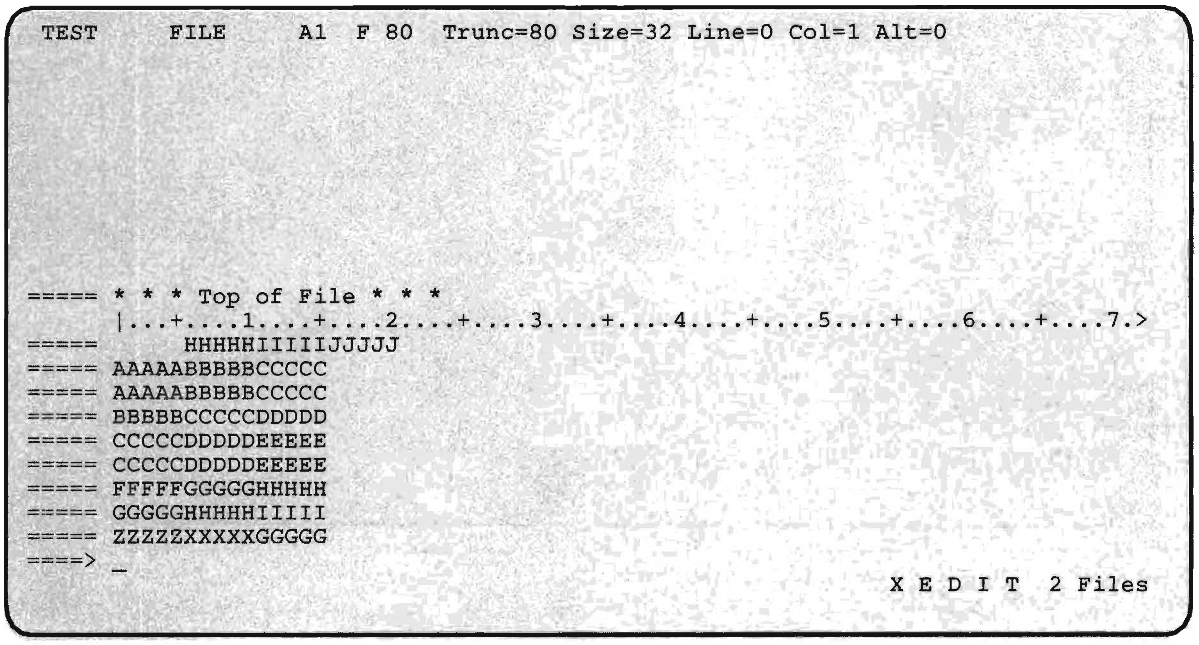
```
 TEST     FILE2    A1  F 80  Trunc=80 Size=32 Line=1 Col=1 Alt=0







===== * * * Top of File * * *
=====      HHHHHIIIIIJJJJJ
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== AAAAABBBBBCCCCC
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== ZZZZZXXXXXGGGGG
===== ZZZZZXXXXXGGGGG
====> _
                                                           X E D I T  2 Files
```

You can switch (or toggle) between the two files by entering the **X** subcommand whenever you want to turn your attention to one file or the other.

Splitting the Screen

It is often useful to be able to examine both files at once. The editor lets you to do this with the **SET SCREEN** subcommand. If you have been experimenting with the **X** subcommand, toggling back and forth between files, use it now if necessary so that TEST FILE A is on the screen.

Your screen should now look like this:

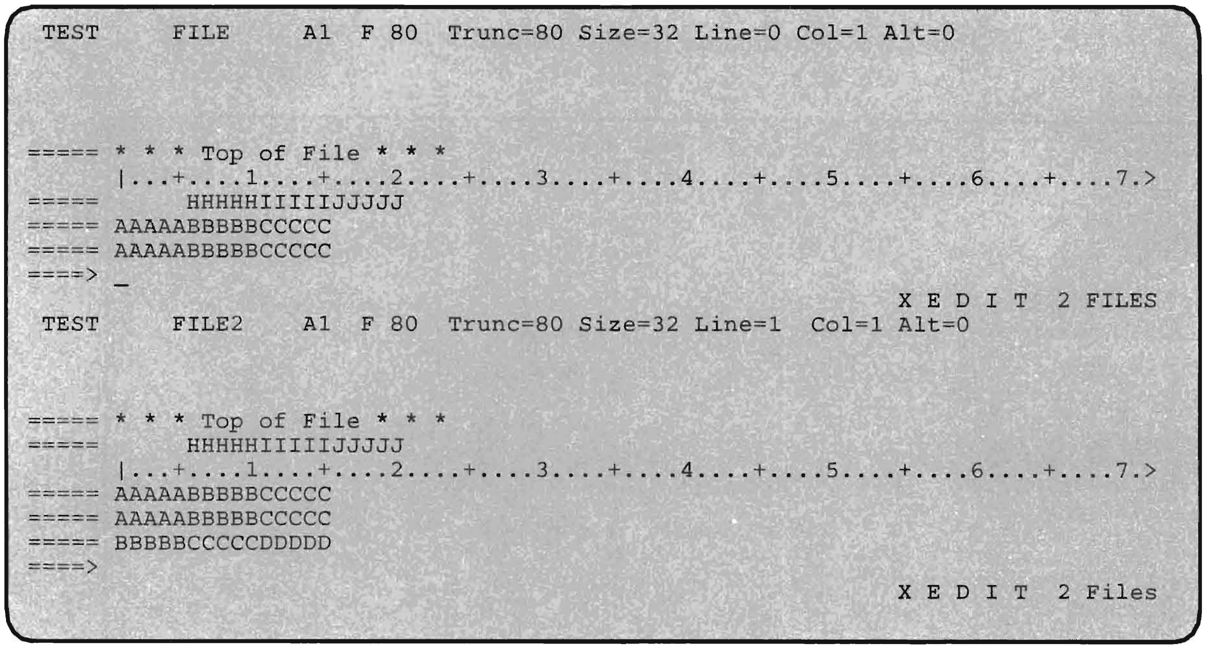


```
 TEST     FILE     A1  F 80  Trunc=80 Size=32 Line=0 Col=1 Alt=0







===== * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
=====      HHHHHIIIIIJJJJJ
===== AAAAABBBBBCCCCC
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== ZZZZZXXXXXGGGGG
====> _
                                                           X E D I T  2 Files
```

Now enter the subcommand:

```
set screen 2
```

Your screen should now look like this:

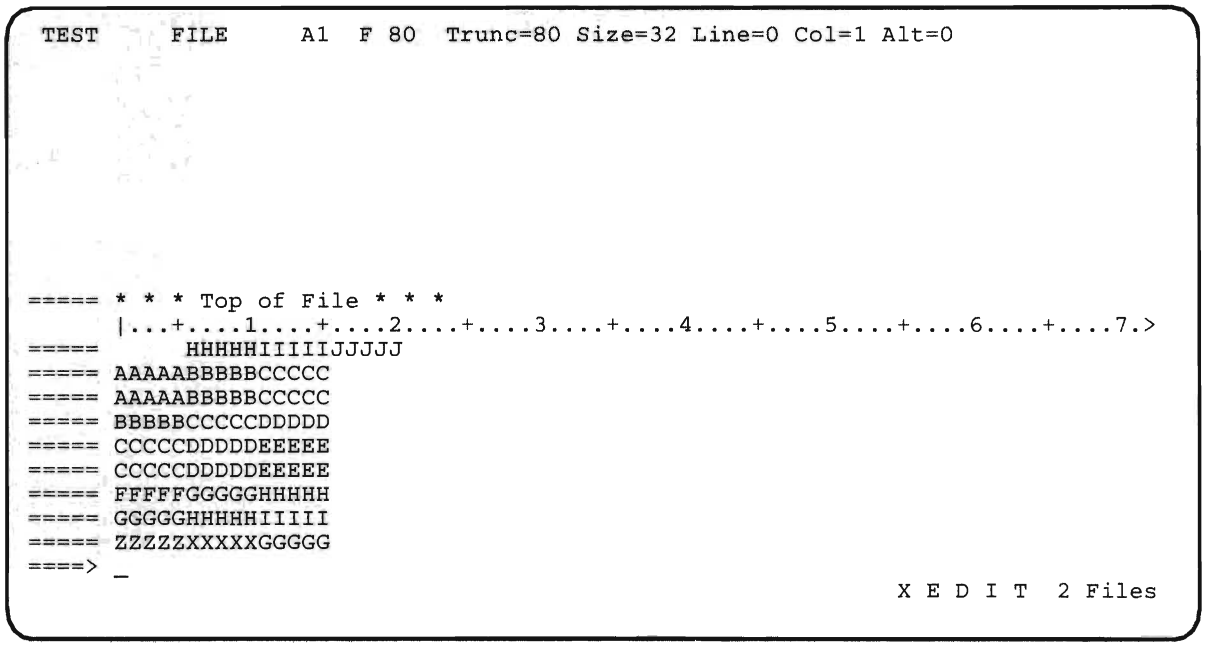


```
 TEST     FILE     A1  F 80  Trunc=80 Size=32 Line=0 Col=1 Alt=0



===== * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
=====      HHHHHIIIIIJJJJJ
===== AAAAABBBBBCCCCC
===== AAAAABBBBBCCCCC
====> _
                                                           X E D I T  2 FILES
 TEST     FILE2    A1  F 80  Trunc=80 Size=32 Line=1  Col=1 Alt=0



===== * * * Top of File * * *
=====      HHHHHIIIIIJJJJJ
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== AAAAABBBBBCCCCC
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
====>
                                                           X E D I T  2 Files
```

If you had entered **SET SCREEN 2** when TEST FILE2 A was on the screen, TEST FILE2 A would be on the top half of the screen and TEST FILE A would be on the bottom half. The **SET** portion of the subcommand isn't required; you could also have enter the subcommand **SCREEN 2**. Now enter the subcommand:

```
screen 1
```

Your screen should now look like this:



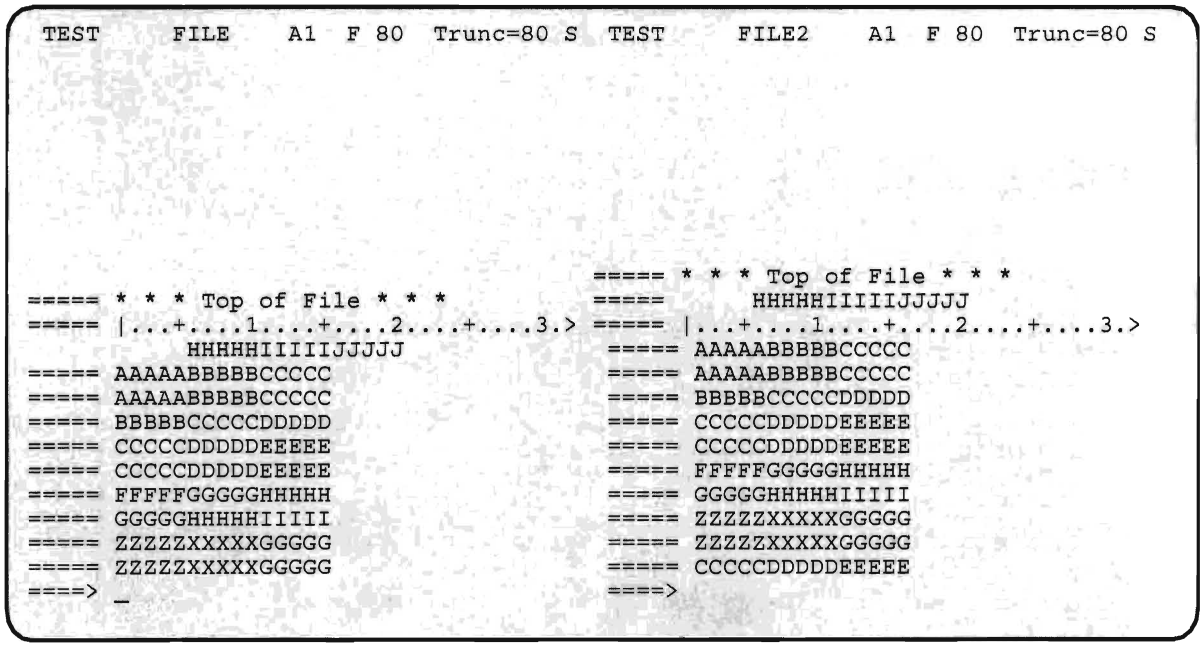
```
 TEST     FILE     A1  F 80  Trunc=80 Size=32 Line=0 Col=1 Alt=0







===== * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
=====      HHHHHIIIIIJJJJJ
===== AAAAABBBBBCCCCC
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== ZZZZZXXXXXGGGGG
====> _
                                                          X E D I T  2 Files
```

If you were editing three files and wanted to see all three of them on the screen, you would enter **SCREEN 3** and so on.

Horizontal screen images are useful in providing complete lines of data for comparison. If the lines of data are very short (as in our examples), you might find it more useful to be able to compare screen images side by side. The editor lets you to do this by inserting the character **V** following the number of screen images. Now enter the subcommand:

```
screen 2 v
```

Your screen should now look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 S TEST     FILE2     A1  F 80  Trunc=80 S







                                       ===== * * * Top of File * * *
===== * * * Top of File * * *          =====     HHHHHIIIIIJJJJJ
===== |...+....1....+....2....+....3.> ===== |...+....1....+....2....+....3.>
          HHHHHIIIIIJJJJJ               ===== AAAAABBBBBCCCCC
===== AAAAABBBBBCCCCC                   ===== AAAAABBBBBCCCCC
===== AAAAABBBBBCCCCC                   ===== BBBBBCCCCCDDDDD
===== BBBBBCCCCCDDDDD                   ===== CCCCCDDDDDEEEEE
===== CCCCCDDDDDEEEEE                   ===== CCCCCDDDDDEEEEE
===== CCCCCDDDDDEEEEE                   ===== FFFFFGGGGGHHHHH
===== FFFFFGGGGGHHHHH                   ===== GGGGGHHHHHIIIII
===== GGGGGHHHHHIIIII                   ===== ZZZZZXXXXXGGGGG
===== ZZZZZXXXXXGGGGG                   ===== ZZZZZXXXXXGGGGG
===== ZZZZZXXXXXGGGGG                   ===== CCCCCDDDDDEEEEE
====> _                                 ====>
```

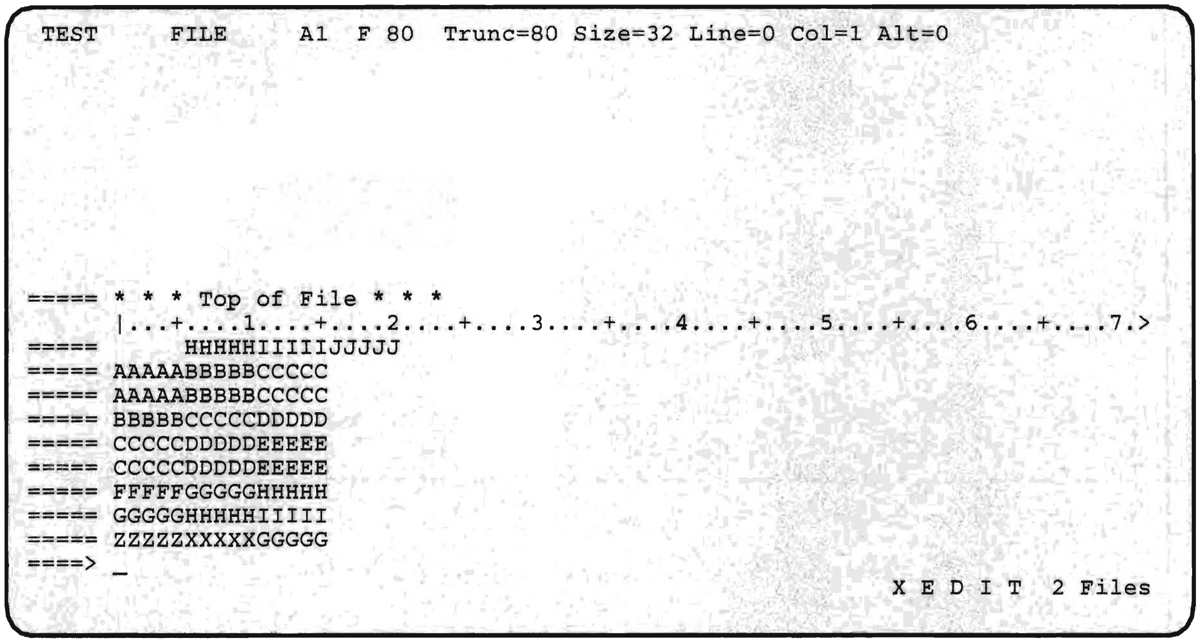
Many people find it easier to compare two files side by side rather than top to bottom.

Each of the displays has its own status line and also its own command line. This is because each file can be manipulated independently of the other.

Before continuing with the next section, return the screen to a single image:

```
screen 1
```

Your screen should now look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=32 Line=0 Col=1 Alt=0






===== * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
=====      HHHHHIIIIIJJJJJ
===== AAAAABBBBBCCCCC
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== ZZZZZXXXXXGGGGG
====> _
                                                          X E D I T  2 Files
```

Using Tabs with the Editor

The **tabulation (tab)** feature lets you align columns accurately and automatically. This is because the normal tab settings for certain filetypes are known to the editor. In this section we'll discuss tab settings and how you can use them to correctly format data in programs, documents, and data files.

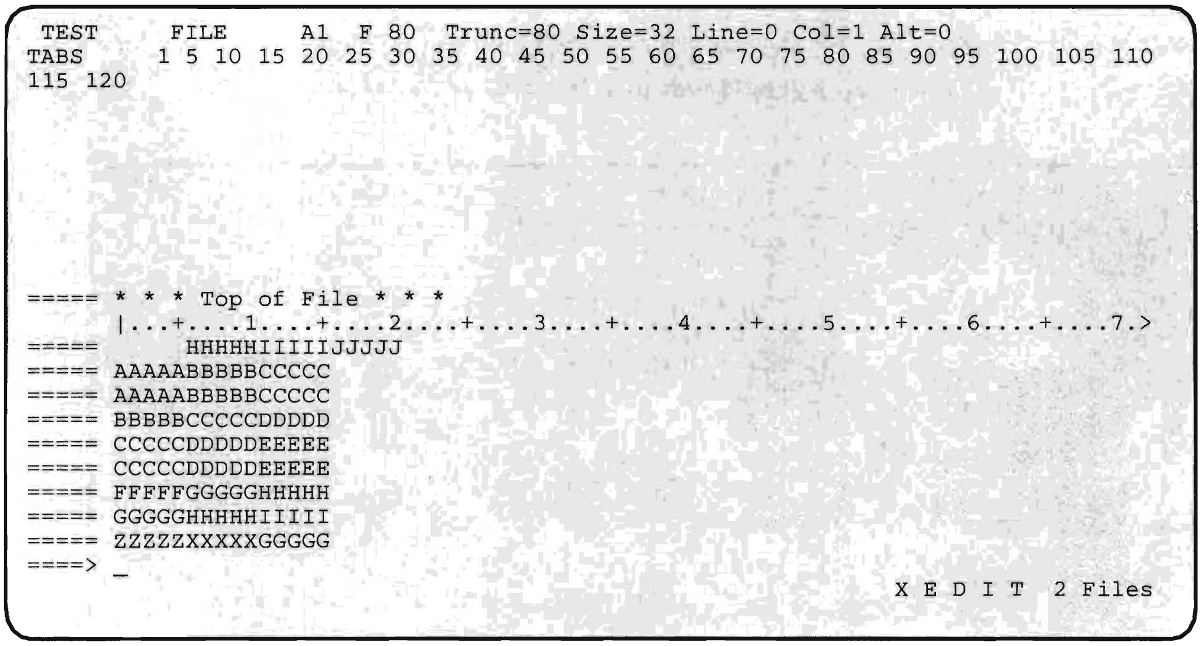
As you may remember, tab settings are linked by the editor to the filetype being edited. If the filetype isn't one of the standard filetypes recognized by the editor, tabs are set as follows:

Tabs	1 5 10 15 20 25 30 ...

To verify this, enter the following subcommand:

```
q tabs
```

Your screen should now look like this:



```
 TEST      FILE      A1  F 80  Trunc=80 Size=32 Line=0 Col=1 Alt=0
TABS      1 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 105 110
115 120









===== * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
=====      HHHHHIIIIIJJJJJ
===== AAAAABBBBBCCCCC
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== ZZZZZXXXXXGGGGG
====> _
                                                          X E D I T  2 Files
```

The tables below indicate default tab settings for these filetypes.

If the filetype is COBOL the tab settings are:

Tabs	1 8 12 20 28 36 44 68 72 80

If the filetype is FORTRAN, the settings are:

Tabs	1 7 10 15 20 25 30 80

If the filetype is FREEFORT the settings are:

Tabs	9 15 18 23 28 33 38 81

Before proceeding to the next session, enter the file subcommand twice:

```
file
file
```

to write the current files to your A-disk.

The QUIT and QQUIT Subcommands

Both the **QUIT** and **QQUIT** subcommands let you end the current editor session and leave the previous copy of the file, if any, intact. The difference between the two subcommands is that **QUIT** only works on an unchanged file, while **QQUIT** works on a file even if it has been changed.

The QUIT Subcommand

If you look at a file but make no modifications to it, you can **QUIT** the editor without updating the disk file. Assume that after looking at the file, you now have the information you want. You don't need to **SAVE** the file since no data has been changed. This is the time to use the **QUIT** subcommand. This terminates the session without saving the file.

The QQUIT Subcommand

Use the **QQUIT** subcommand during an edit session if you've made a mistake and want to recover the original version of the file.

Let's assume that you want to delete 2 lines but make an error and enter **d22** (as you know, **D** means delete.). In this case you have inadvertently deleted 22 lines of data. Clearly we don't want to **FILE** or **SAVE** this data. However, if you try to **QUIT** the file, the subcommand is rejected. Your screen appears with a message:

```
   File has been changed; use QQUIT to quit anyway
```

This is because **QUIT** only functions on an unchanged file. Since our present file has been changed, and the change was unintentional, we should use **QQUIT**. With **QQUIT**, you can leave the session on a changed file, but the file won't be written to disk. When you enter **QQUIT** the session is terminated. Be sure you understand the difference between **QUIT** and **QQUIT**.

Ways to End an Editing Session

The table below indicates five ways to end an editing session:

Subcommand	Leave Edit	Write to Disk	Remarks
FILE	yes	yes	all changes saved
SAVE	no	yes	all changes saved
QUIT	yes	no	if no changes made
QQUIT	yes	no	unconditional
CANCEL	yes	no	if no changes made

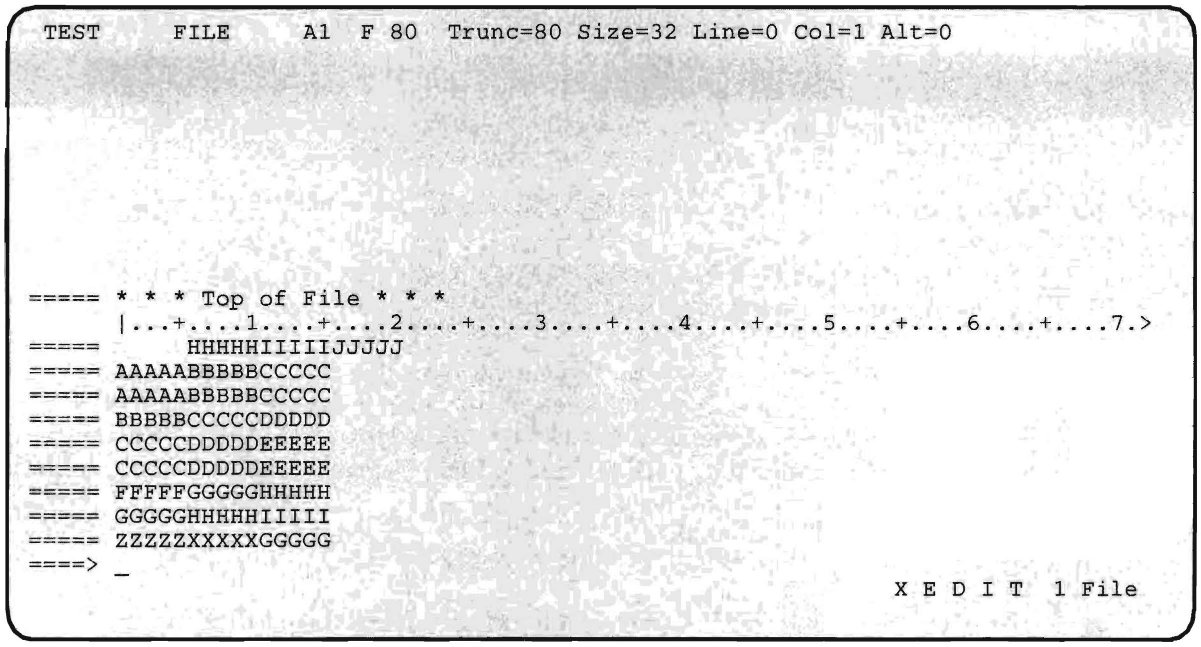
The **CANCEL** subcommand can be used to terminate the edit session on multiple files.

Using the AUTOSAVE Function

The **AUTOSAVE** function lets you specify how often the editor should write a copy of the file you are editing to a special work file on disk. First, begin a new edit session by entering:

```
x test file (noprof
```

Your screen should look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=32 Line=0 Col=1 Alt=0






===== * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
=====      HHHHHIIIIIJJJJJ
===== AAAAABBBBBCCCCC
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== ZZZZZXXXXXGGGGG
====> _
                                                           X E D I T  1 File
```

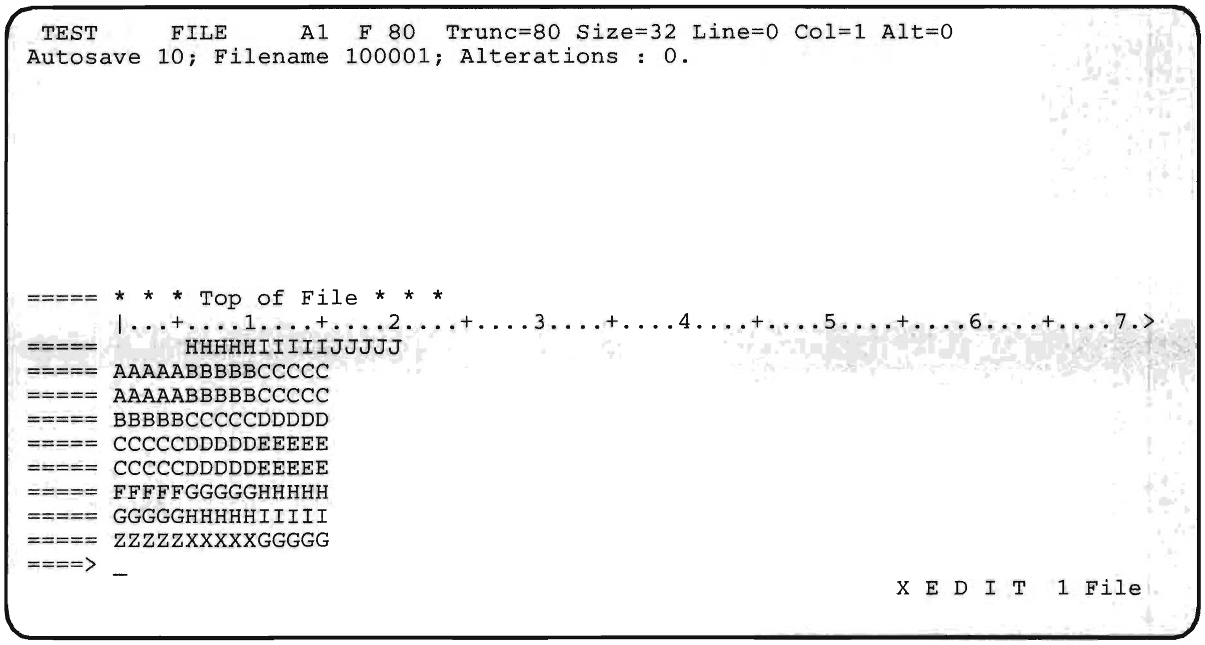
Now set the autosave function by entering the subcommand:

```
set autosave 10
```

No message is returned. Confirm the setting by entering the subcommand:

```
query autosave
```

Your screen should now look like this:



```
 TEST     FILE     A1  F 80  Trunc=80 Size=32 Line=0 Col=1 Alt=0
Autosave 10; Filename 100001; Alterations : 0.








===== * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
=====      HHHHHIIIIIJJJJJ
===== AAAAABBBBBCCCCC
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== ZZZZZXXXXXGGGGG
====> _
                                                          X E D I T  1 File
```

The elements of the autosave status message are:

Autosave 10	means that the file is saved as a special autosave file after ten or more modifications of a certain magnitude (for example, a line added, changed, or a GET subcommand issued) have been made.
Filename n	is the file name of the autosave file, assigned by the editor for this session. The file type is AUTOSAVE and the file mode is A.
Alterations: 0	means that no alterations have been made since the last autosave.

Insert five lines to follow the top of the file. Do this by entering the prefix subcommand **A5** on the Top of File line. Press ENTER to put the cursor on the command line.

Enter the subcommand:

```
query autosave
```

The alterations field now reads 1. The autosave function considers adding lines one alteration. Return the cursor to the added lines and key in the letter **A** on each line. Don't press the ENTER key until all five lines have the letter **A** on them. Again, enter the subcommand:

```
query autosave
```

The alterations field now reads 6. A changed line counts as one alteration, even if the ENTER key isn't pressed for each line. Now change the first three lines by entering the subcommands

```
next
c//ggggg/3
```

on the command line. Issue the QUERY AUTOSAVE subcommand again.

The alterations field now contains 7.

If you delete a block of lines or use GET to copy several lines, each action counts as one alteration. Enter the following subcommands, one after the other:

```
next
del 2
query autosave
```

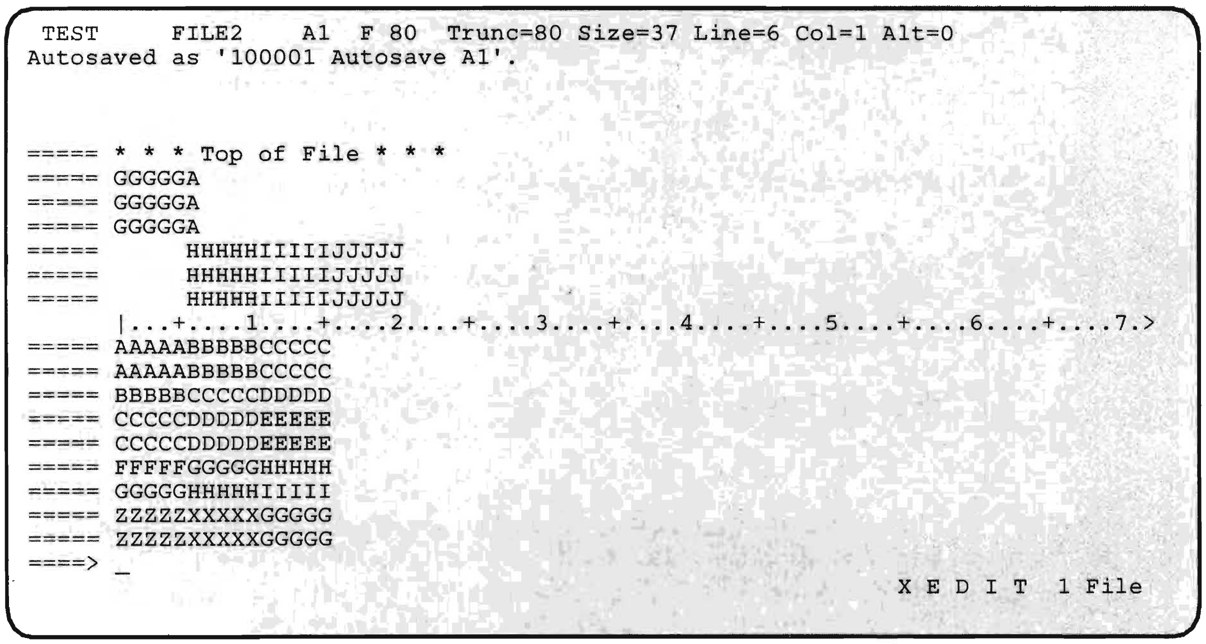
The alterations field now contains 8. Now enter the subcommands:

```
dup
query autosave
```

The alterations field now contains 9. Following the next command that changes the file, the editor will **AUTOSAVE** the file. Enter the following subcommand:

```
dup
```

Your screen should now look like this:



```
 TEST     FILE2    A1  F 80  Trunc=80 Size=37 Line=6 Col=1 Alt=0
Autosaved as '100001 Autosave A1'.



===== * * * Top of File * * *
===== GGGGGA
===== GGGGGA
===== GGGGGA
=====      HHHHHIIIIIJJJJJ
=====      HHHHHIIIIIJJJJJ
=====      HHHHHIIIIIJJJJJ
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
===== AAAAABBBBBCCCCC
===== AAAAABBBBBCCCCC
===== BBBBBCCCCCDDDDD
===== CCCCCDDDDDEEEEE
===== CCCCCDDDDDEEEEE
===== FFFFFGGGGGHHHHH
===== GGGGGHHHHHIIIII
===== ZZZZZXXXXXGGGGG
===== ZZZZZXXXXXGGGGG
====> _
                                                          X E D I T  1 File
```

The file has been automatically saved. The name of the autosave file is shown on the message line.

Note: The autosave file is erased when a FILE is done to the source file.

You can use autosave to recover lost work. You can use the autosave file as you would any CMS file. You can edit it, or rename it by using the **RENAME** command in CMS, and then work with the renamed file. The alterations number (specified when you set autosave on) limits the alterations that can be lost. If you set the alterations number to 1, the file is saved after any alteration.

Using the CMS Update Facility

Another feature of the editor is the **UPDATE** option, which makes use of the CMS update facility to control changes made to your files.

The **UPDATE** option is useful because:

- The original source file isn't directly changed.
- All changes are recorded in separate update files.
- The changes are time-stamped and identified.
- Changes can be applied and removed on a selective and controlled basis.
- A separate log is automatically generated to maintain a record of the changes.

Let's return to the sample program that you created in the previous chapter. Enter one of the two following CMS commands:

```
x testprog cobol (update noprof
```

or

```
x testprog fortran (update noprof
```

depending on whether the program you created in the previous chapter was written in COBOL or FORTRAN.

The result of specifying the update option is the following:

1. The editor gets the source file TESTPROG COBOL or TESTPROG FORTRAN.
2. The editor looks for an updated file called TESTPROG UPDATE.
3. If the editor can't find TESTPROG UPDATE, it creates TESTPROG UPDATE and displays the contents of TESTPROG COBOL or TESTPROG FORTRAN on the screen, but with the file name of

TESTPROG UPDATE at the top of the screen. Any changes that are to be made to the source are now recorded in the file TESTPROG UPDATE for later application to the source file with the **CMS UPDATE** command.

4. If the editor does find a file called TESTPROG UPDATE, it gets the file and applies all updates recorded in it to TESTPROG COBOL or TESTPROG FORTRAN. It then displays the updated file on the screen. New updates are recorded in the TESTPROG UPDATE file.

The Update File

The update file consists of two types of records:

- New or changed source statements
- Update control statements.

Some update control statements are automatically generated by the editor, while others must be entered manually.

Three kinds of update control statements are generated by the editor:

Statement	**Type**
./ I	Insert
./ D	Delete
./ R	Replace

These three control statements, along with the appropriate new or changed source statements, are recorded in the update file by the editor when you're updating a source program. Each update control statement also carries a time and date stamp in columns 52 through 71, reflecting when you created or changed the update statement.

The layout of the control cards is as follows:

Columns	**Contents**
1-2	./
4	I, D, or R
6-13	Sequence number of source statement
24	$ (or other delimiter)
26-29	Starting statement number value, if this card applies to more than one statement in the source file.

Columns	Contents
31-33	Incrementing statement number value, if this card applies to more than one statement in the source file.

The following two update control statements are never automatically generated by the update facility. If you use them, add them manually into the update file using the editor.

Statement	Type
./ S	Cause Resequencing
./ *	Comment

Resequencing affects columns 73 through 80. It should be added to the update file using the editor if you want the source sequence to be resequenced. **./ *** is a **comment** statement using the editor. You can add comments to document the updates.

The UPDATE Command

You use the **UPDATE** command (UPDATE fn ft) to apply update changes from the update file to the source file. Four files are involved in the process: two for input, with which you're already familiar, and two output files.

The input files are the source file (our example, TESTPROG COBOL or TESTPROG FORTRAN) and the update file (our example, TESTPROG UPDATE).

Two files are created by the update process. They're an updated source file, $TESTPRO COBOL or $TESTPRO FORTRAN, and an update log file, TESTPROG UPDLOG.

Note: When the update facility creates a workfile, it uses the filename of the source file, and prefixes it with the dollar sign **($)**. Since the filename can only be 8 characters long, this means that the last character of an 8-character source file name will be dropped from the workfile. This is why the workfile for the TESTPROG filename is called $TESTPRO.

The file $TESTPRO COBOL or $TESTPRO FORTRAN is the updated source for your next compilation. File TESTPROG UPDLOG has a detailed record of the updates to the TESTPROG source. The leading **$** in the $TESTPRO COBOL or $TESTPRO FORTRAN file name is used to indicate that the file has been created using the **UPDATE** command.

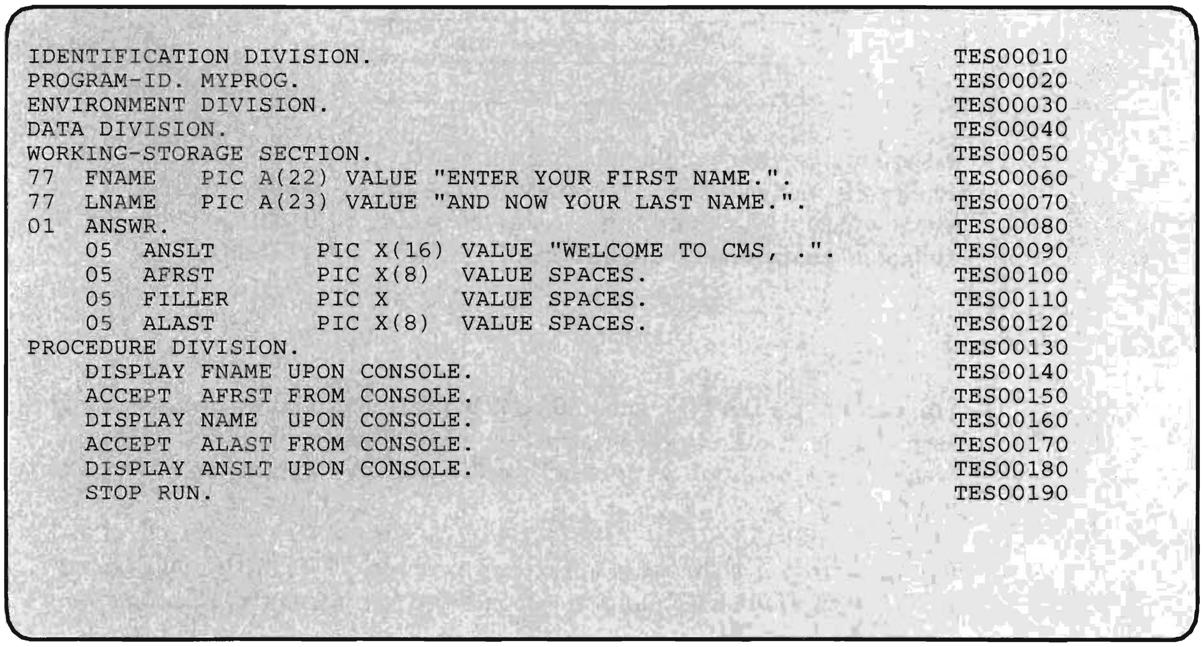
If you're programming in FORTRAN, skip to "Updating a FORTRAN Source File" on page 100.

Updating a COBOL Source File

The following is an example using the COBOL source file you created in the previous chapter. Now enter the command:

```
type testprog cobol
```

The result should look like this:



```
IDENTIFICATION DIVISION.                                      TES00010
PROGRAM-ID. MYPROG.                                           TES00020
ENVIRONMENT DIVISION.                                         TES00030
DATA DIVISION.                                                TES00040
WORKING-STORAGE SECTION.                                      TES00050
77  FNAME   PIC A(22) VALUE "ENTER YOUR FIRST NAME.".         TES00060
77  LNAME   PIC A(23) VALUE "AND NOW YOUR LAST NAME.".        TES00070
01  ANSWR.                                                    TES00080
    05  ANSLT       PIC X(16) VALUE "WELCOME TO CMS, .".      TES00090
    05  AFRST       PIC X(8)  VALUE SPACES.                   TES00100
    05  FILLER      PIC X     VALUE SPACES.                   TES00110
    05  ALAST       PIC X(8)  VALUE SPACES.                   TES00120
PROCEDURE DIVISION.                                           TES00130
    DISPLAY FNAME UPON CONSOLE.                               TES00140
    ACCEPT  AFRST FROM CONSOLE.                               TES00150
    DISPLAY NAME  UPON CONSOLE.                               TES00160
    ACCEPT  ALAST FROM CONSOLE.                               TES00170
    DISPLAY ANSLT UPON CONSOLE.                               TES00180
    STOP RUN.                                                 TES00190
```

There are sequence numbers in columns 73 through 80. These were automatically generated by the editor. Each sequence number is prefixed with the letters TES, the first three letters of the filename. The editor generates the sequence numbers in this form because the default file characteristics are in effect:

- TRUNC was set to 72, allowing serialization in columns 73 to 80.
- SERIAL was set to ON 10 10, meaning that:
  - The first three positions of the sequence number would be filled with the first three characters of the filename.
  - The numerical portion of the sequence number (columns 76 through 80) would begin with 10.
  - Each subsequent sequence number would be incremented by 10.

To make use of the update option of the editor, you'll have to convert the sequence numbers to all numerics - that is, all eight characters of the sequence number must be used. The editor makes this task very simple. First, bring the program into the editor by entering the command:

```
x testprog cobol (noprof
```

Now enter the following subcommands, one after the other:

```
serial all 10 10
file
```

Serial ALL means that all eight characters of the sequence field are used for numeric sequencing. Type the file on the terminal again using the **TYPE** command. The file should look like this:

```
IDENTIFICATION DIVISION.                                                00000010
PROGRAM-ID. MYPROG.                                                     00000020
ENVIRONMENT DIVISION.                                                   00000030
DATA DIVISION.                                                          00000040
WORKING-STORAGE SECTION.                                                00000050
77  FNAME   PIC A(22) VALUE "ENTER YOUR FIRST NAME.".                   00000060
77  LNAME   PIC A(23) VALUE "AND NOW YOUR LAST NAME.".                  00000070
01  ANSWR.                                                              00000080
    05  ANSLT       PIC X(16) VALUE "WELCOME TO CMS, .".                00000090
    05  AFRST       PIC X(8)  VALUE SPACES.                             00000100
    05  FILLER      PIC X     VALUE SPACES.                             00000110
    05  ALAST       PIC X(8)  VALUE SPACES.                             00000120
PROCEDURE DIVISION.                                                     00000130
    DISPLAY FNAME UPON CONSOLE.                                         00000140
    ACCEPT  AFRST FROM CONSOLE.                                         00000150
    DISPLAY NAME  UPON CONSOLE.                                         00000160
    ACCEPT  ALAST FROM CONSOLE.                                         00000170
    DISPLAY ANSLT UPON CONSOLE.                                         00000180
    STOP RUN.                                                           00000190
```

Now the sequence numbers are all numeric.

Now we'll make these changes:

- Add a line between the second and third record of the file (that is, between the PROGRAM-ID statement and the ENVIRONMENT DIVISION statement. This line will contain the AUTHOR statement.
- Move line 6 to follow line 7.

First, call the editor with the update option using the command:

```
x testprog cobol (update noprof
```

and notice that the filetype is now UPDATE. We're now ready to make the changes indicated above.

1. Position the cursor in the prefix area of line 2 (the PROGRAM-ID statement), and enter:

   ```
   ==a==
   ```

   to add a new line. The cursor is now at the beginning of the new line.

2. Since the AUTHOR statement should begin in column 12, press the **PF4** key (the TAB key) twice to position the cursor in column 12.

3. Now type in the AUTHOR line:

   ```
   author. sam jones.
   ```

4. Now move the cursor down to the seventh line, which reads:

   ```
   77  FNAME   PIC A(22) VALUE IS "ENTER YOUR FIRST NAME.".
   ```

   and enter the following prefix subcommand:

   ```
   m====
   ```

   but don't press ENTER. We'll need a target to move the line to, so move the cursor to the next line, which reads:

   ```
   77  LNAME   PIC A(23) VALUE IS "AND NOW YOUR LAST NAME.".
   ```

   and enter the following prefix subcommand:

   ```
   f====
   ```

   and press ENTER. The result will be that lines 7 and 8 have swapped position. We have now made the changes described above. Press ENTER.

5. Now, close out the editing session by entering the command:

   ```
   file
   ```

If you now type the original program, TESTPROG COBOL, you'll see that none of the changes you made have taken place in the source file. This is because the update option has created a new file called TESTPROG UPDATE, which contains the changes you made, together with the control statements necessary to implement the changes again. Now type the TESTPROG UPDATE file. It should look like this:

```
./ I 00000020        $ 25 5                          03/08/84 11:24:11
       AUTHOR. SAM JONES.
./ D 00000060                                        03/08/84 11:24:11
./ I 00000070        $ 75 5                          03/08/84 11:24:11
       77  FNAME   PIC X(22) VALUE "ENTER YOUR FIRST NAME.".
       00000060
```

Note: The date and time stamp values reflect the time you issued the **file** subcommand.

Now we'll use the CMS UPDATE command and to update the source. Enter the following CMS command:

```
update testprog cobol
```

to update the source and create a new file called $TESTPRO COBOL and an update log called TESPTOR UPDLOG. The source file $TESTPRO COBOL should look like this:

```
IDENTIFICATION DIVISION.                                            00000010
PROGRAM-ID. MYPROG.                                                 00000020
AUTHOR. SAM JONES.                                                  ********
ENVIRONMENT DIVISION.                                               00000030
DATA DIVISION.                                                      00000040
WORKING-STORAGE SECTION.                                            00000050
77  LNAME   PIC X(23) VALUE "AND NOW YOUR LAST NAME.".              00000070
77  FNAME   PIC X(22) VALUE "ENTER YOUR FIRST NAME.".               ********
01  ANSWR.                                                          00000080
    05  ANSLT       PIC X(16) VALUE "WELCOME TO CMS, .".            00000090
    05  AFRST       PIC X(8)  VALUE SPACES.                         00000100
    05  FILLER      PIC X     VALUE SPACES.                         00000110
    05  ALAST       PIC X(8)  VALUE SPACES.                         00000120
PROCEDURE DIVISION.                                                 00000130
    DISPLAY FNAME UPON CONSOLE.                                     00000140
    ACCEPT  AFRST FROM CONSOLE.                                     00000150
    DISPLAY NAME  UPON CONSOLE.                                     00000160
    ACCEPT  ALAST FROM CONSOLE.                                     00000170
    DISPLAY ANSLT UPON CONSOLE.                                     00000180
    STOP RUN.                                                       00000190
```

The update log file TESTPROG UPDLOG should look like this:

```
1UPDATING 'TESTPROG COBOL    A1' WITH 'TESTPROG UPDATE   A1'           UPDATE
LOG -- PAGE       1
0       ./ I 00000020         $ 25 5                          03/08/84 11:24:11
 INSERTING...             AUTHOR. SAM JONES.
         ********
        ./ D 00000060                                         03/08/84 11:24:11
 DELETING...              77  FNAME   PIC X(22) VALUE "ENTER YOUR FIRST NAME.".
         00000060
        ./ I 00000070         $ 75 5                          03/08/84 11:24:11
 INSERTING...             77  FNAME   PIC X(22) VALUE "ENTER YOUR FIRST NAME.".
         ********
```

For more information about the update option, see the *VM/SP CMS User's Guide.*

If you're programming in COBOL, skip to "Chapter 4: More about Compiling and Running a Program" on page 105.

Updating a FORTRAN Source File

The following is an example using the FORTRAN source file you created in the previous chapter. Now enter the command:

```
type testprog fortran
```

The result should look like this:

```
      PROGRAM MYPROG                                                    TES00010
      CHARACTER*8 F,S                                                   TES00020
      WRITE (6,5)                                                       TES00030
      READ (5,2) F                                                      TES00040
      WRITE (6,10)                                                      TES00050
      READ (5,2) S                                                      TES00060
      WRITE (6,15) F,S                                                  TES00070
2     FORMAT (A8)                                                       TES00080
5     FORMAT (' ENTER YOUR FIRST NAME.')                                TES00090
10    FORMAT (' AND NOW YOUR LAST NAME.')                               TES00100
15    FORMAT (' WELCOME TO CMS, ',A8,1X,A8)                             TES00110
      STOP                                                              TES00120
      END                                                               TES00130
```

There are sequence numbers in columns 73 through 80, which were automatically generated by the editor. Each sequence number is prefixed with the letters TES, the first three letters of the filename. The editor generated the sequence numbers in this form because the default file characteristics were in effect:

- TRUNC was set to 72, allowing serialization in columns 73 to 80.
- SERIAL was set to ON 10 10, meaning that:
  - The first three positions of the sequence number would be filled with the first three characters of the filename.
  - The numerical portion of the sequence number (columns 76 through 80) would begin with 10.
  - Each subsequent sequence number would be incremented by 10.

In order to make use of the update option of the editor, you'll have to convert the sequence numbers to all numerics — that is, all eight characters of the sequence number must be used. The editor makes this task very simple. First, bring the program into the editor by entering the command:

```
x testprog fortran (noprof
```

Now enter the following subcommands, one after the other:

```
serial all 10 10
file
```

SERIAL ALL means that all eight characters of the sequence field will be used for numeric sequencing. Type the file on the terminal again using the **TYPE** command. The file should look like this:

```
      PROGRAM MYPROG                                            00000010
      CHARACTER*8 F,S                                           00000020
      WRITE (6,5)                                               00000030
      READ (5,2) F                                              00000040
      WRITE (6,10)                                              00000050
      READ (5,2) S                                              00000060
      WRITE (6,15) F,S                                          00000070
2     FORMAT (A8)                                               00000080
5     FORMAT (' ENTER YOUR FIRST NAME.')                        00000090
10    FORMAT (' AND NOW YOUR LAST NAME.')                       00000100
15    FORMAT (' WELCOME TO CMS, ',A8,1X,A8)                     00000110
      STOP                                                      00000120
      END                                                       00000130
```

Now the sequence numbers are all numeric.

Now we'll make these changes:

- Add a line between the second and third record of the file (that is, between the CHARACTER*8 statement and the WRITE (6,5) statement. This line will contain a comment line giving the programmer's name.
- Move line 9 to follow line 10.

First, call the editor with the update option, using the command:

```
x testprog fortran (update noprof
```

and notice that the filetype is now UPDATE. We're now ready to make the changes indicated above.

1. Now position the cursor in the prefix area of line 2 (the CHARACTER*8 statement), and enter:

   ```
   ==a==
   ```

   to add a new line. The cursor is now at the beginning of the new line.

2. Since the comment must begin in column 1, you can now type the comment line:

   ```
   c author. sam jones.
   ```

3. Press ENTER and issue the DOWN 5 command. Now move the cursor to the tenth line, which reads:

   ```
   5      FORMAT (' ENTER YOUR FIRST NAME.')
   ```

   and enter the following prefix subcommand:

   ```
   m====
   ```

but don't press ENTER. We'll need a target to move the line to, so move the cursor to the next line, which reads:

```
10     FORMAT (' AND NOW YOUR LAST NAME.')
```

and enter the following prefix subcommand:

```
f====
```

and press ENTER. The result will be that lines 9 and 10 have swapped position. We've now made the changes described above.

4. Press ENTER. Now, close out the editing session by entering the command:

```
file
```

If you now type the original program, TESTPROG FORTRAN, you'll see that none of the changes you made have taken place in the source file. This is because the update option has created a new file called TESTPROG UPDATE, which contains the changes you made, together with the control statements necessary to implement the changes again. Now type the TESTPROG UPDATE file. It should look like this:

```
./ I 00000020          $ 25 5                          03/08/84 11:24:11
C AUTHOR. SAM JONES
./ D 00000090                                          03/08/84 11:24:11
./ I 00000100          $ 105 5                         03/08/84 11:24:11
5      FORMAT (' ENTER YOUR FIRST NAME.')
       00000090
```

Note: The date and time stamp values reflect the time you issued the **file** subcommand.

Now we'll use the **UPDATE** command and to update the source. Enter the following command:

```
update testprog fortran
```

to update the source and create a new file called $TESTPRO FORTRAN and an update log called TESPTOR UPDLOG. The source file $TESTPRO FORTRAN should look like this:

```
      PROGRAM MYPROG                                                    00000010
      CHARACTER*8 F,S                                                   00000020
C AUTHOR. SAM JONES.                                                    ********
      WRITE (6,5)                                                       00000030
      READ (5,2) F                                                      00000040
      WRITE (6,10)                                                      00000050
      READ (5,2) S                                                      00000060
      WRITE (6,15) F,S                                                  00000070
2     FORMAT (A8)                                                       00000080
10    FORMAT (' AND NOW YOUR LAST NAME.')                               00000100
5     FORMAT (' ENTER YOUR FIRST NAME.')                                ********
15    FORMAT (' WELCOME TO CMS, ',A8,1X,A8)                             00000110
      STOP                                                              00000120
      END                                                               00000130
```

The update log file TESTPROG UPDLOG should look like this:

```
1UPDATING 'TESTPROG FORTRAN    A1' WITH 'TESTPROG UPDATE   A1'          UPDATE
LOG -- PAGE       1
0       ./ I 00000020          $ 25 5                          03/08/84 11:24:11
INSERTING...    C AUTHOR. SAM JONES.
        ********
        ./ D 00000090                                          03/08/84 11:24:11
DELETING...     5     FORMAT (' ENTER YOUR FIRST NAME.')
           00000090
        ./ I 00000100          $ 105 5                         03/08/84 11:24:11
INSERTING...    5     FORMAT (' ENTER YOUR FIRST NAME.')
        ********
```

This completes the example of how to use the update facility in CMS.

For more information about the update option, see the *VM/SP CMS User's Guide*.

Summary

In this chapter we've discussed, in some detail, how to use the System Product Editor, including how to use the EDIT and INPUT mode, how to manipulate data and the display of data, and how to use the CMS Update Facility.

Chapter 4: More about Compiling and Running a Program

This chapter provides more information about compiling, loading, and executing COBOL and FORTRAN programs under VM. It discusses various VM commands that are useful and necessary for running your programs. It also tells you how to run multiple object modules.

Note: If you're programming in FORTRAN, skip to "More about FORTRAN Compilers" on page 107.

Files Created by the COBOL Compiler

While executing, the COBOL compiler makes use of unused space on your A-disk for its intermediate workfiles. These files are assigned a filename equal to that of the source program, with filetypes of SYSUT1, SYSUT2, SYSUT3, SYSUT4, and SYSUT6. (The SYSUT5 filetype is produced when the SYMDMP option is specified on the command line.) These workfiles (except SYSUT5) are erased at the normal end of compilation. If the compiler is halted prematurely, some or all of these files may still reside on the A-disk. If this is the case, you should erase them before invoking the compiler again.

Figure 7 on page 106 illustrates what happens in your system.

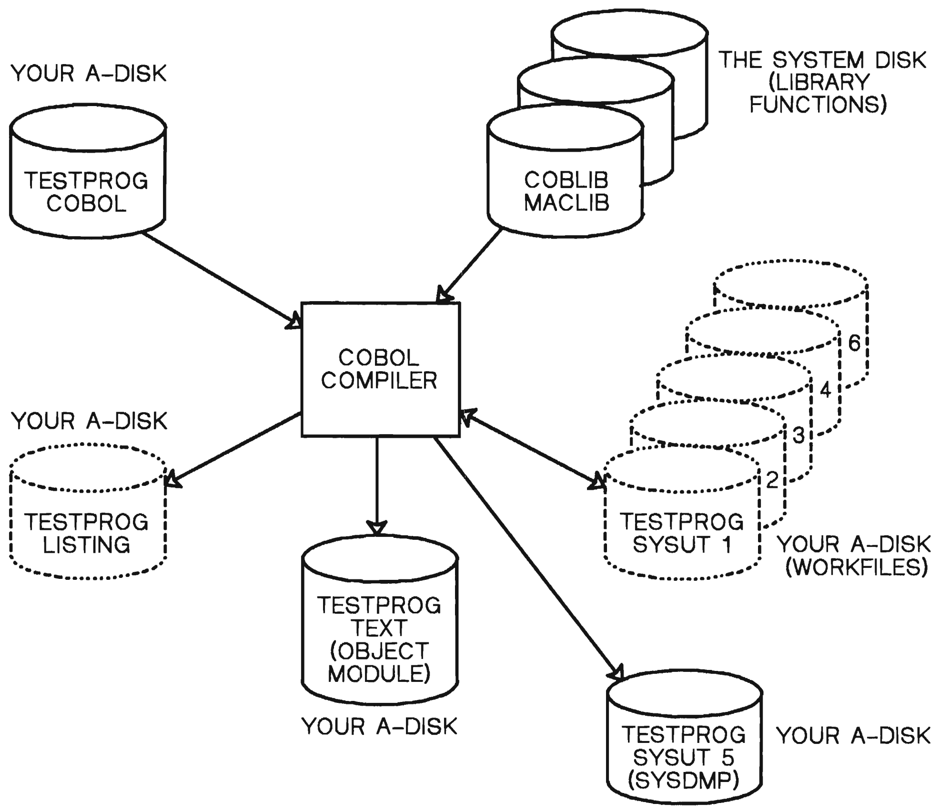


Figure 7. Files Used by the COBOL Compiler

When the compiler writes an output disk file, it's placed on a read/write CMS disk. This is usually your A-disk. However, if the source program is on another disk also accessed in read/write mode, the compiler output files are written to that disk. If the source program is on a read-only disk that is an extension of a read/write disk (for example, a disk accessed as C/B), the files are written to that read/write disk (in this case, the B-disk). If the read-only disk is not an extension of another disk, the files are written to the first read/write disk in the CMS search order.

TEXT files contain the machine-language object code generated by the COBOL compiler. In addition to TEXT files, the compiler produces a compilation listing for each COBOL source file that you compile. The listing is placed on your A-disk (unless LISTING was assigned to some other read/write disk) in a file with a filetype of LISTING.

Depending on how your installation is set up, you may get some or all of the following:

- A formatted source listing.
- Diagnostic messages.
- Cross references (if the XREF option is used).

- A glossary, global tables, literal pools, and register assignments (if the DMAP option is used).
- Global tables, literal pools, register assignments, and assembler language expansion of the source program (if the PMAP option is used).
- A condensed listing of the compiler generated object code (if the CLIST option is used).

More about FORTRAN Compilers

In "Chapter 3: Using the System Product Editor" on page 39, we used the VS FORTRAN Version 2 compiler to compile your program. It's one of several FORTRAN compilers available under VM.

The VS FORTRAN Compiler

You can invoke the VS FORTRAN Version 2 compiler and library in CMS with the **FORTVS2** command.

The VS FORTRAN Version 2 Compiler and Library program product is designed according to the specification of the following industry standards, as understood and interpreted by IBM as of May, 1982.

The following two standards are technically equivalent.

- American National Standard Programming Language FORTRAN, ANSI X3.9-1978 (also known as FORTRAN 77)
- International Organization for Standardization ISO 1539-1980 Programming Language-FORTRAN

The following two standards are technically equivalent.

- American Standard FORTRAN, X3.9-1966 (also known as FORTRAN 66)
- International Organization for Standardization ISO R (1539-1972 Programming Languages-FORTRAN)

IBM has implemented FORTRAN 77 and FORTRAN 66 with IBM extensions.

It is especially useful for scientific and engineering applications involving mathematical computations and other manipulations of numeric data.

Some features are:

- VSAM sequential and direct file processing through VSAM data sets.

- More flexible and direct control of character variables and arrays through the character data type.
- Structured programming aids, such as the block IF statement.
- Symbolic dumps of variables at abnormal termination.

Figure 8 illustrates what happens in your system.

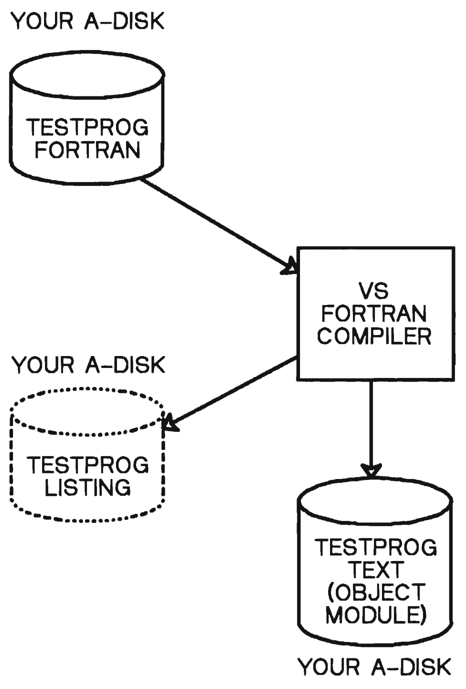


Figure 8. Files Used by the FORTRAN Compiler

When the compiler writes an output disk file, it's placed on a read/write CMS disk. This is usually your A-disk. However, if the source program is on another disk also accessed in read/write mode, the compiler output files are written to that disk. If the source program is on a read-only disk that is an extension of a read/write disk (for example, a disk accessed as C/B), then the files are written to that read/write disk (in this case, the B-disk). If the read-only disk is not an extension of another disk, the files are written to the first read/write disk in the CMS search order.

The LISTING file contains the compilation listing for the source file you're compiling.

Depending on how your installation is set up, you may get some or all of the following:

- A compilation listing for the source file you're compiling.
- Diagnostic messages.
- Cross references (if the XREF option is used).

- Storage map (if the MAP option is used).

Copying OS Files from CMS MACLIBS

If you use the OS COBOL or VS FORTRAN Version 2 compiler COPY statement, and have frequently-used files that contain COBOL and FORTRAN code, you can place them in a CMS file called MACLIB. Then, you can identify the MACLIB to be searched during compilation.

Use the **GLOBAL** or **FILEDEF** command to make the MACLIB available to the COBOL and FORTRAN compiler. You can use the following commands, for example, to make a library called COPYLIB MACLIB available to the compiler:

```
global maclib copylib
```

The **GLOBAL** command identifies the library to be searched during the compilation. The **FILEDEF** command connects the CMS filename to the compiler ddname SYSLIB and points to the medium (disk) on which the file resides. You can also use multiple MACLIBs, as well as PDS libraries on OS disks, concatenating these libraries under the single ddname SYSLIB. See "Using Macro Libraries" on page 119 for more information.

Defining Input and Output Files

When you execute an OS program under CMS that has input or output files, you must first identify the files to CMS with the **FILEDEF** command. The **FILEDEF** command in CMS performs the same functions as the data definition (DD) card in OS JCL: it describes the input and output files.

When you enter the **FILEDEF** command, you specify the following information:

- The **ddname**.
- The **device type**.
- A **file identifier**, if the device type is DISK.
- The **type of label** on your tape file, if tape label processing is specified.
- One or more **options**, as necessary.

The **FILEDEF** command connects the **logical I/O control statements (LIOCS)** in your program with the **physical I/O control statements (PIOCS)** that define the input/output files outside the program.

If you're programming in FORTRAN, skip to "Specifying the DDNAME in FORTRAN" on page 110.

Specifying the DDNAME in COBOL

If you're writing a COBOL program, the ddname is specified with the **SELECT** or **ASSIGN** clause in the **FILE CONTROL** paragraph. For example, your program could contain the following FILE CONTROL paragraph and FD statements:

```
FILE-CONTROL.
      SELECT INFILE ASSIGN TO UR-3505-S-CARDIN.
      SELECT OUTFILE ASSIGN TO DA-3330-S-OUTDD.
       .
       .
       .
FD    INFILE
       .
       .
       .
FD    OUTFILE
       .
       .
       .
```

In this case the ddname for infile is **cardin**, and the ddname for outfile is **outdd**. These are the names you would use in the ddname portion of the **FILEDEF** command.

Note: If you're programming in COBOL, skip to "Specifying the Device Type" on page 111.

Specifying the DDNAME in FORTRAN

In FORTRAN, (if there is no OPEN statement or if the OPEN statement does not specify a ddname) the ddname has the following default format:

```
FTxxFyyy
```

where:

xx	is the FORTRAN data set reference number specified in an I/O verb (READ, WRITE) in your program.
yyy	is a sequence number (from 001 to 999) that identifies multiple files under the same data set reference numbers. For direct access files, this number is always 001. For sequential files, the number varies depending on the order in which the file is referred to in your program.

For example, if your FORTRAN program contains the following I/O statements:

```
      WRITE (6,10)
   10 FORMAT (' A=?')
      READ (5,20) A
   20 FORMAT (F8.3)
      A=A**2
      WRITE (6,10) A
   30 FORMAT ('A=',F8.3)
```

the ddname for the file referenced by the data set reference number 5 (defined in the READ statement) would be:

```
FT05F001
```

The ddname for data set reference number 6 (defined in the WRITE statements) would be:

```
FT06F001
```

Data set reference number **5** causes a read from the keyboard; data set reference number **6** causes a write to the screen unless you issue a FILEDEF to override these default assignments.

See *VS FORTRAN Version 2 Application Programming: Guide* for more information.

Specifying the Device Type

For input files, the device type you enter on the FILEDEF command line indicates the device from which you want records read. The device type can be:

DISK	for files on CMS or OS disks.
TERMINAL	for keyboard input.
READER	for input from your virtual reader.
TAPn	for tape. The **n** is used to designate multiple tape drives.

For output files, the device you specify can be:

DISK	for files on CMS or OS disks.
TERMINAL	for terminal output.
PRINTER	for the virtual printer.
TAPn	for tape. The **n** is used to designate multiple tape drives.
PUNCH	for the virtual punch.

Sometimes you'll specify a FILEDEF, but won't need the output. This happens, for example, when you want to test a program, and are concerned more with its logic than with the correctness of the output. In this case, you can specify the device type DUMMY. FILEDEF will provide the necessary linkage between LIOCS and PIOCS, but no actual data will be produced. You can also use the DUMMY device type for input FILEDEFs if your program can run without input.

Specifying CMS Files for Input and Output

Sometimes you need to specify a CMS file identifier as part of the device type specification. Do this if the device type you specify is DISK, and the input or output file is to be on a CMS disk. For example, suppose your TESTPROG COBOL program has an input ddname of TDATA, and this file resides on your A-disk in a file called TEST DATA A1. You'd enter the following command to identify the file to CMS prior to running your program:

```
filedef tdata disk test data
```

For a program written in FORTRAN, the ddname might be FT05F001, if the data set reference number 5 is associated with a READ statement. In this case, the command:

```
filedef ft05f001 disk test data
```

or

```
filedef 05 disk test data
```

enables the program to read data from the CMS file TEST DATA A1.

Specifying FILEDEF Options

The FILEDEF command provides several options to control file specifications:

- BLOCK (or BLOCKSIZE), LRECL, RECFM, and DSORG specify and describe the file format and organization.
- PERM specifies whether the file definition is to be permanent (until CHANGE, or IPL (next session)).
- MEMBER specifies if the file is a member of an OS partitioned data set or CMS MACLIB or TXTLIB.
- You can specify whether output is to be in upper case or mixed case.
- There are several controls for tape I/O.

See *VM/SP CMS Command Reference* for a complete description of each option.

Loading Object Modules

When you've issued the necessary FILEDEFs to resolve LIOCS/PIOCS linkage, you can issue the **LOAD** and **END** commands (or the **LOAD** command with the **START**). If you have multiple modules to be loaded, you may need to issue the **INCLUDE** command. The CMS loader loads files into storage as a result of the **LOAD** and **INCLUDE** commands, as shown below.

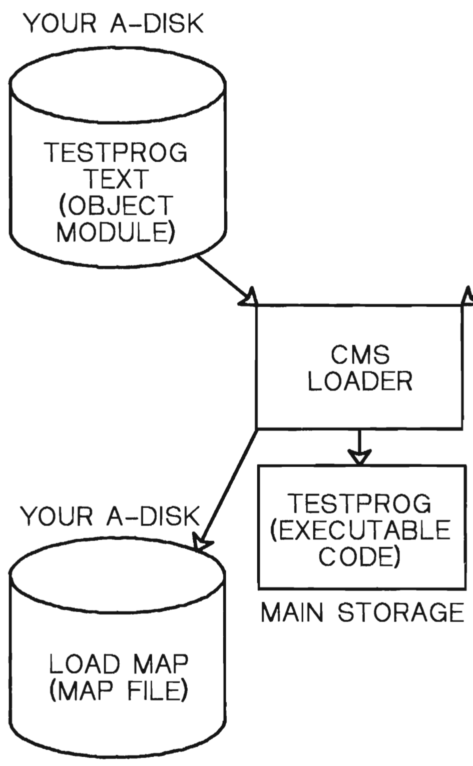


Figure 9. The CMS Loader

When a file is loaded, the loader checks for unresolved references. If there are any (as a result of a CALL to a subprogram, for example), the loader searches your disks for TEXT files with filenames that match the external entry name. When it finds a match, it loads the TEXT file into storage. The loader searches your A-disk for a file called GETDATE TEXT. As soon as it finds the first file with that filename and filetype, the loader loads the file and resolves references between the two programs.

If the loader can't find a file, it issues a message followed by a list of unresolved entry point names. This occurs, for example, if the CSECT name for a module is different from the CMS filename. In this case, you can use the **INCLUDE** command to resolve the reference.

For example, suppose that TESTPROG issues a call to a subprogram named GETDATE. Suppose that GETDATE exists with a CMS file identifier of DATERTN TEXT A1. After you issue the command:

```
load testprog
```

the loader issues the message:

```
   THE FOLLOWING NAMES ARE UNDEFINED:
    GETDATE
Ready;
```

You issue the following command to resolve the reference:

```
include datertn
```

The **INCLUDE** command has the same format and option list (with one exception) as the **LOAD** command. The main difference between them is that when you issue the **INCLUDE** command, the loader tables aren't reset.

If you issue two **LOAD** commands in succession, the second **LOAD** command cancels the effect of the first, and the pointers to the files are lost.

You can specify as many **INCLUDE** commands as necessary to load files into storage. Don't use the **INCLUDE** command unless you've just issued a **LOAD** command.

You can issue a **GLOBAL** command between **LOAD** and **INCLUDE** (or between two **INCLUDEs**) if a TXTLIB is to be searched for unresolved entry points.

When the **LOAD** and **INCLUDE** commands execute, they produce a **load map**. The load map indicates entry points loaded and their virtual storage locations. You may find the load map useful in debugging your programs. If you don't specify the NOMAP option, the load map is written onto your A-disk as LOAD MAP A5. Each time you issue the **LOAD** command, the old load map is replaced by a new one.

In addition to the options provided by the **LOAD** and **INCLUDE** commands, you can also use **loader control statements**. You can insert these statements in TEXT files using the editor. These statements allow you to:

- Set the location counter to control the load address of the next TEXT file.
- Modify instructions and constants in a TEXT file (patch a program).
- Change the entry point.
- Nullify an external reference.

See *VM/SP CMS command Reference* for a description of these statements (as well as the standard loader statements produced by the compiler).

Determining Program Entry Points

When you load a single TEXT file into storage, the default entry point is the first CSECT name loaded in the object file. Use the RESET option (on either the **LOAD** or **INCLUDE** command line) to specify a different entry point at which to start execution.

For example, if TESTPROG has an entry point TEST02, start execution at that point with the command line:

```
load testprog (reset test02 start
```

If you don't specify an entry point, the loader searches for an entry point in the following order, selecting the first line it finds:

1. An entry point specified on the command line.
2. The last entry point specified with the RESET option on a **LOAD** or **INCLUDE** command.
3. The name on the last ENTRY statement read by the loader.
4. The name on the last LDT statement that contained an entry name read by the loader. The LDT statement is produced by the compiler.
5. The name on the first compiler-produced END statement read by the loader.
6. The first byte of the first CSECT loaded.

Issuing Dynamic Loads with OS Macros

An area of concern to OS programmers is resolution of external references when various OS macros are used for dynamic loading (LINK, LOAD, or XCTL macros). If you use these to call members of CMS TXTLIBs (see "The CMS Loadlib" on page 133), the CMS loader determines the entry point of the called program and returns the entry point to your program.

If you load a TXTLIB member that has a VCON to another TXTLIB member, the LDT card from the second member may be the last LDT card read by the loader. If this LDT card specifies the name of the second member, CMS may return that entry point address to your program, rather than the address of the first member.

Summary

This chapter discussed compiling, loading, and executing COBOL and FORTRAN programs under VM. It described some VM commands that are useful and necessary for running your programs, and told you how to run multiple object modules.

Chapter 5: Using CMS Libraries

All the CMS library types have a similar structure. Each one contains one or more members and has an **internal directory**. The library facilities use this directory to locate members. Since libraries are unlike other CMS files, you create, update, and use them differently than you do other CMS files. The three types of libraries are:

1. **Macro** Files (of filetype MACLIB) that contain one or more macros written in assembler language, or copy files written in other languages. These files are referenced when you invoke either the Assembler or one of the compilers to process a program. Some MACLIBs are provided with the COBOL or FORTRAN compilers and contain subroutines used during the compilation process. By using the **MACLIB** command, you can create or change the contents of MACLIBs. You can use the **GLOBAL** command to define a MACLIB; this tells the compiler where to find routines needed when processing source code.

2. **Text or Program** Files (of filetype TXTLIB) created and altered by the **TXTLIB** command. These files are defined for the compiler and linkage editor or loader by the **GLOBAL** command. They are libraries of code already compiled or assembled for use during program execution. As with MACLIBs, some TXTLIBs with run-time subroutines are provided with the COBOL and FORTRAN compilers. You can also create your own TXTLIBs with subroutines or entire programs written for use in one or more applications.

3. **Load** Files (of filetype LOADLIB) that contain **absolute** or **core image** modules, compiled and link-edited. LOADLIBs differ from TXTLIBs, which contain **relocatable** code that is not yet link-edited.

Figure 10 on page 118 shows the relationships among the various filetypes and the System Product Editor, the compilers, and the linkage editor or loader.

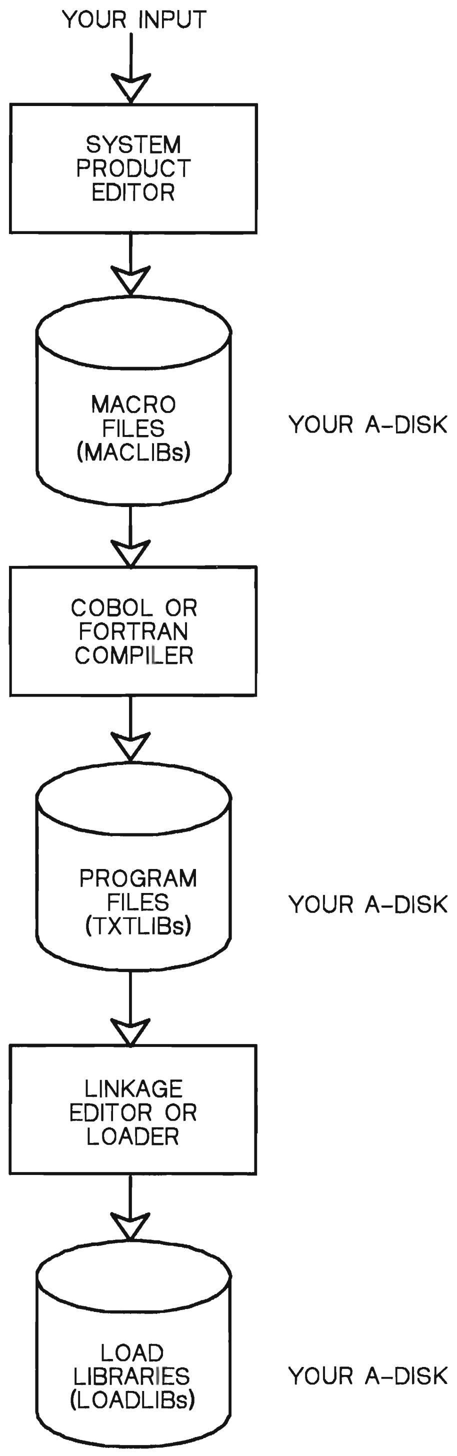


Figure 10. CMS Libraries

This chapter examines the structure of macro and text libraries. Load libraries are discussed in "The LOADLIB Command" on page 135.

Using Macro Libraries

A CMS macro library has a filetype of MACLIB. You can create a MACLIB from files with MACRO and COPY filetypes by using the **MACLIB** command. When used in an assembler language program, macro definitions in a MACRO file generate code **in line** by referencing the macro name. A COPY file contains predefined source statements that are included in a source program when the COPY statement is encountered.

A MACLIB is similar to an OS PDS. It has individual members, which you can create using the editor, or which you can copy from other source files. For a member to be added to a MACLIB, it must be in a CMS file with a filetype of COPY or MACRO. Use the filetype COPY for files containing source code to be included in a MACLIB. The filetype MACRO is usually used for Assembler Language macros.

Internally a CMS MACLIB consists of three parts:

1. The **LIBPDS Statement** - contains information about the library itself.

2. The **Library Members** - separated from one another by a delimiter.

3. The **Directory** - contains pointers to each of the library members.

Here's an example of how a MACLIB called TESTMAC is represented internally:

```
LIBPDS
TEST1     MACRO
          .
          .
          .
          MEND
/ /
TEST2     MACRO
          .
          .
          .
          MEND
/ /
TEST3     MACRO
          .
          .
          .
          MEND
/ /
TEST4     DSECT
          .
          .
          .
/ /
TEST1               TEST2               TEST3               TEST4
```

In this example, the TESTMAC MACLIB contains four members: three macros (TEST1, TEST2, and TEST3) and a COPY file that contains a DSECT program called **TEST4**.

You use the MACLIB command to:

- Create a macro library
- Add or delete members
- List or compress the members in a library.

Creating a New MACLIB

The **GEN** parameter of the MACLIB command generates a CMS macro library from input files specified on the command line. The input files must have a filetype of either MACRO or COPY.

For example, if you want to create the TESTMAC MACLIB used above, enter the command:

```
maclib gen testmac test1 test2 test3 test4
```

This creates a macro library with the file identifier TESTMAC MACLIB A1 from the following files:

```
TEST1 MACRO A1
TEST2 MACRO A1
TEST3 MACRO A1
TEST4 COPY  A1
```

Note: If a library named TESTMAC MACLIB A1 already exists, it's replaced by this new library.

When macros are in a macro library, the name of the library member is taken from the macro prototype statement in the file. If a file contains more than one macro, the **MACLIB** command gets the library member names from the macro prototype statements of each macro in the file. For example, suppose that several macro definitions, including one for TEST3 MACRO, are in the TEST1 MACRO file.

The file might look like this:

```
TEST1     MACRO
          .
          .
          .
          MEND
TEST1A    MACRO
          .
          .
          .
          MEND
TEST3     MACRO
          .
          .
          .
          MEND
```

If you create the TESTMAC MACLIB given in the example, the library now has the following members in this order:

```
TEST1       From TEST1 MACRO A1
TEST1A      From TEST1 MACRO A1
TEST3       From TEST1 MACRO A1
TEST2       From TEST2 MACRO A1
TEST3       From TEST3 MACRO A1
TEST4       From TEST4 COPY A1
```

The TEST3 macro, which appears in both the TEST1 and the TEST3 MACRO files, now exists as two members in TESTMAC MACLIB. But there's only one entry in the MACLIB directory. The **MACLIB** command doesn't check for duplicate macro names. Later, when a program requests TEST3 macro from TESTMAC MACLIB, it uses the first TEST3 macro it meets (from the TEST1 MACRO file).

Adding, Deleting, and Replacing Members

The ADD function of the MACLIB command adds members to a macro library. No checking is done for duplicate names, entry points, or CSECTs. The new member is added at the end of the library.

Suppose you want to add TEST5 COPY to a TESTMAC MACLIB. The command for this action looks like this:

```
 maclib add testmac test5
```

TESTMAC MACLIB now contains the following members:

```
TEST1       From TEST1 MACRO A1
TEST1A      From TEST1 MACRO A1
TEST3       From TEST1 MACRO A1
TEST2       From TEST2 MACRO A1
TEST3       From TEST3 MACRO A1
TEST4       From TEST4 COPY A1
TEST4A      From TEST4 COPY A1
TEST5       From TEST5 COPY A1
```

The REP function replaces members in a macro library by deleting the directory entry for the macro definition in the specified library. It adds new macro definitions to the library and creates new directory entries.

Suppose you want to replace the TEST2 macro with a later debugged version or one with new features or code. The command line:

```
maclib rep testmac test2
```

causes the following actions:

1. The latest version of the TEST2 macro (in the file TEST2 MACRO A1) is added to the library.

2. The old directory entry for the last version of TEST2 is deleted from the library.

3. A new directory entry is created.

The **physical order** of members in the library is arranged so that the new version of TEST2 appears after the old version. The **logical order** (the one in which requests for macros are satisfied) is determined by the directory entry — not by the physical position of the member in the library. The REP function causes the directory entry rather than the source code to be replaced.

The DEL function deletes members from a macro library. What it does is remove the member name from the library directory so there are no unused entries. The macro or copy code still takes up space in the library but can't be accessed because it's been deleted in the directory entry.

Deleting the last remaining member of a MACLIB erases the entire MACLIB.

If a library contains two members with the same name, only the first member is deleted from the directory.

Suppose you create TESTMAC MACLIB with the command:

```
maclib gen testmac test1 test2 test3 test4
```

Next you replace TEST2 MACRO using:

```
maclib rep testmac test2
```

Now you want to back out (that is, make unavailable) the first version of TEST3 macro, the one from the TEST1 file. You do this by using the command:

```
maclib del testmac test3
```

The result is this:

```
 TEST1     From TEST1 MACRO A1
 TEST1A    From TEST1 MACRO A1
(TEST3     From TEST1: present but unavailable)
(TEST2     From TEST2: present but replaced)
 TEST3     From TEST3 MACRO A1
 TEST4     From TEST4 COPY A1
 TEST4A    From TEST4 COPY A1
 TEST2     From TEST2 MACRO A1, later version
```

If you have MACRO and COPY files (on any accessed disk) with the same filename, the MACRO version is used when you invoke the MACLIB command.

Compressing a MACLIB

When you use the ADD, DEL, and REP functions repeatedly, the library ends up with "dead entries" or "nonmembers." These are macros and copy code that remain in the library but are no longer used since they have no directory entries. You can use the COMP function to compress a library by deleting any macros or copy blocks that don't have directory entries.

The **MACLIB** command does this by copying each member of the file to a new file, using the directory. The new file now has the temporary name of MACLIB CMSUT1. This name is always used, regardless of the original macro library filename. After all valid library members are copied to MACLIB CMSUT1, the old library is erased and the temporary CMSUT1 file is renamed with the old library name.

To continue our example, the results above show that TESTMAC MACLIB now contains two nonmembers. One is the TEST2 macro that was replaced by a later version. The other is the TEST3 macro that was deleted. To save DASD space, you may want to compress TESTMAC to eliminate the two nonmembers. You issue the command:

```
maclib comp testmac
```

The resulting library contains the same valid members as those listed above. However, the ones in parentheses (the first version of TEST3 and the earlier version of TEST2) no longer occupy space in the MACLIB. Thus, the new TESTMAC MACLIB is smaller than the old one. It lost the two files plus two delimiter records. The directory size remains the same, since it was already compressed. The result is this:

```
TEST1      From TEST1 MACRO A1
TEST1A     From TEST1 MACRO A1
TEST3      From TEST3 MACRO A1
TEST4      From TEST4 COPY A1
TEST4A     From TEST4 COPY A1
TEST2      From TEST2 MACRO A1
```

Examining Contents of a MACLIB

You can use the MAP function to list certain information about members in a macro library. This information includes:

- Member name
- Size of the member
- Sequential position in the library.

You can obtain this information in three different ways:

- As a file on your A-disk (the DISK option, the default)
- As a file and as a spooled printer file (the PRINT option)

- As a display on your terminal (the TERM option).

The DISK and PRINT options create a file with the filetype MAP and filemode A5. The filename is the same as the MACLIB being mapped. All three options erase any existing MAP file for the specified MACLIB.

Using CMS Commands to Manipulate Members

The macro library facilities in CMS include a number of CMS commands that can address particular members of a MACLIB. By using these commands, you can manipulate MACLIB members without altering the MACLIB itself.

These commands can address MACLIB members by name:

FILEDEF	sets up a file definition for a member.
PRINT	prints a MACLIB member.
TYPE	displays a member on the terminal.

The **MACLIST** command displays a list of information about all members in a specified macro library. **MACLIST** provides you with an easy way to select and edit CMS maclib members. CMS commands can be issued against the members directly from the displayed list. The commands execute when you press the enter key (which is set to the **EXECUTE** command).

In the **MACLIST** environment, information that is normally provided by the **MACLIB** command (with the **MAP** option) is displayed under the control of the System Product editor. You can use **XEDIT** subcommands to manipulate the list itself. See the *CMS Command Reference* for more information on the **MACLIST** command.

You can also use the **MOVEFILE** command with an appropriate FILEDEF to extract a member from a library. The MACLIB member you specify is copied directly from the MACLIB to your A-disk.

Extracting a Maclib Member

If you copy a member from a given MACLIB onto your A-disk (for example, to make changes to it), you can use the MOVEFILE command. You must first issue a file definition for:

- The member name that is input to the **MOVEFILE** command
- The output file that is written to your A-disk

Let's say you want to make some changes to TEST DSECT in our example of TESTMAC MACLIB. When you added TEST to the TESTMAC

MACLIB, you may have erased the source copy to save some disk space. So, the original is no longer available to you.

The following command sequence extracts the TEST DSECT from TESTMAC MACLIB. It then copies it to your A-disk with the file identifier of TEST COPY A1:

```
filedef inmove disk testmac maclib (member test
filedef outmove disk test copy a1
movefile
```

Now you can edit TEST COPY and make the changes you want. Then you can do a MACLIB REP to replace TEST in TESTMAC MACLIB.

Note: All CMS files you created by this method include the MACLIB delimiter statement / / as the last record in the file. So the first change you should make to a MACLIB member extracted in this way is to delete this / / delimiter record.

More about MOVEFILE

The **MOVEFILE** command in the example above is a simple application that makes use of the existing FILEDEFs. But with the **PDS** option, you can use MOVEFILE to extract every member of a macro library.

For example:

```
filedef test1 disk testmac maclib a
filedef macro disk
movefile test1 macro (pds
```

This sequence defines TESTMAC MACLIB as the input file for the **MOVEFILE** command, and assigns a temporary logical name of **test1** to the file. The second **FILEDEF** command identifies the filetype of the resulting files. It specifies that they're to be written to disk. The **MOVEFILE** command then causes **test1** (that is, TESTMAC MACLIB A1) to be moved into separate files with a filetype of macro.

Note: Each member in this example has a filetype of MACRO, including those with the original filetype of COPY. You must rename those back to their original filetype of COPY by using the **CMS RENAME** command. Each CMS file resulting from a **MOVEFILE** command from a MACLIB has the delimiter (/ /) statement as the last record of the file. To replace any member of the library, first delete the delimiter record from the input MACRO or COPY file.

Printing and Displaying MACLIB Members

The **PRINT** and **TYPE** commands both accept the option MEMBER as a means of specifying a single MACLIB member, or all the members. The format of these commands is similar.

For example, this command line causes TEST1 to be printed:

```
print testmac maclib (member test1
```

If you code the MEMBER option with an asterisk (*), all the members are printed.

If you enter the following command line:

```
type testmac maclib (member *
```

all the members of MACLIB are displayed.

One spool file is produced for each member printed. To print all the members of a library continuously without separator pages between them, issue the **SPOOL PRINT CONT** command. Then, if you want to return to printing files with separator pages between them, use the **SPOOL PRINT NOCONT CLOSE** command.

System MACLIBs

So far we've looked at macro libraries as your own private libraries. But some macro libraries are supplied as part of the CMS system. These contain various CMS and OS Assembler Language macros that you may want to use in your programs.

Two of the system macro libraries supplied are specific to the CMS environment. They contain macros used by CMS itself. Many of these are useful in manipulating CMS files or in displaying data or messages on the terminal.

These libraries are:

CMSLIB MACLIB contains CMS macros from VM/370.

DMSSP MACLIB contains macros that are new or changed in VM/SP.

When assembling programs that use CMS macros, be sure to specify these libraries in the **GLOBAL** command. On the command line, use DMSSP before CMSLIB.

```
global maclib dmssp cmslib
```

Two other system MACLIBs contain OS macros:

OSMACRO MACLIB contains OS macros that CMS supports or simulates or those that require no CMS support.

OSMACRO1 MACLIB contains macros that CMS doesn't support or simulate.

You can assemble programs in CMS that contain these macros. However, they can only be executed in OS, not in CMS.

Two system MACLIBs support subsets of specific OS functions:

OSVSAM MACLIB contains the subset of supported OS/VSAM macros.

TSOMAC MACLIB contains TSO macros.

To get a list of any of these library macros, use the MAP function of the **MACLIB** command.

Text Libraries

TXTLIBs contain relocatable object modules that can be referenced in two ways:

- You can use CMS commands such as **LOAD** and **INCLUDE** to create nonrelocatable modules.
- Programs can reference TXTLIBs at run time.

TXTLIBs, like MACLIBs, have directories and members. You create them by using the **TXTLIB** command. The **TXTLIB** command has a similar format to the **MACLIB** command, except for the absence of the REP and COMP functions:

GEN	creates the TXTLIB.
ADD	adds members to the TXTLIB.

The total number of members in any given TXTLIB can't exceed 1000. When this number is reached, an error message is displayed.

The total number of entry points in members can't exceed 254 if the filename option is specified, 255 if not. An error message is displayed when this limit is reached and processing has begun on a new file. When processing terminates, the TXTLIB created includes all the text files entered up to, but not including, the one that caused the overflow.

DEL	deletes members from the TXTLIB. If you delete the last remaining member of a TXTLIB, the TXTLIB is erased.
MAP	lists the members of the TXTLIB.

Since there is no REP function, you must use the DEL function followed by ADD to replace an existing TXTLIB member. Each function is discussed in greater detail below.

When a TEXT file is added to a library, its membername or membernames are taken from the CSECT names or statements in the TEXT file unless the filename option is specified. If the filename option is specified, the membername is equal to the filename. Deletions and **LOAD** and **INCLUDE** command references must be made on these membernames; they may be different from the CMS filename from which they originated.

If the FILename option is specified and you have a TEXT file with the filename TESTPROG and a CSECT named CHECK, when you issue the **TXTLIB** command:

```
txtlib add testlib testprog (filename
```

the TESTLIB TXTLIB has a new member called TESTPROG with CHECK as an entry point in that member.

If the FILename option is not specified and you have a TEXT file with the FILename TESTPROG and a CSECT named CHECK, when you issue the **TXTLIB** command:

```
txtlib add testlib tesprog
```

the TESTPROG TXTLIB has a new member called CHECK.

You must delete members by their initial entry in the dictionary (that is, their **name** or the first ID name). Any attempt to delete a specific alias or entry point within a member results in a **NOT FOUND** message.

The internal structure of a TXTLIB is similar to that of a MACLIB:

```
LIBPDS
 ESD            TESTPRG1                               00000001
 TXT                                                   00000002
 END                          15741SC103 020183297 00000003
 LDT
/  / LDT
LIBPDS
 ESD            TESTPRG2                               00000001
 TXT                                                   00000002
 END                          15741SC103 020183297 00000003
 LDT
/  / LDT
LIBPDS
 ESD            TESTPRG3                               00000001
 TXT                                                   00000002
 END                          15741SC103 020183297 00000003
 LDT
/  / LDT
TESTPRG1        TESTPRG2        TESTPRG3
```

In this example, the TXTLIB (which we'll call TSTLIB TXTLIB A1) has three members: TESTPRG1, TESTPRG2, and TESTPRG3. The delimiter for TXTLIB has the additional characters LDT following the / / characters.

Its function is to separate library members from one another: it works in the same way as the MACLIB delimiter. The last records in the file are the directory, which has the same structure as the MACLIB directory.

The members of a TXTLIB consist of files with a filetype of TEXT. These are generated from assemblies and compilations. When you compile a COBOL or FORTRAN program, the resulting relocatable object module is given:

- a filename corresponding to the source program
- a filetype of TEXT.

The TEXT file can't be executed directly because it's relocatable; the addresses are all relative to location zero. This is the standard form for all assembler and compiler output. Each TEXT file is made up of at least one each of the following types of records:

ESD	is an **External Symbol Dictionary** statement. This is the first statement in the module (and therefore the first statement in each member of a TXTLIB). The ESD statement contains the name of the entry point (CSECT) of the module.
TXT	is a statement that contains the actual machine code of the program generated by the assembler or the compiler.
LDT	is a **Loader Termination** statement. It contains data required by the loader program when the module is loaded into storage before execution or the creation of a nonrelocatable module.
TXTLIB	functions operate in a similar way to MACLIB functions:
GEN	creates a TXTLIB on your A-disk. If a TXTLIB with the same name already exists, it's replaced by the new one. The filename option can be used on this version of the TXTLIB command.
ADD	adds TEXT files to the end of an existing TXTLIB on a read/write disk. No checking is done for duplicate names, entry points, or CSECTs. The filename option can be used on this version of the TXTLIB command.
FILENAME	indicates that all of the filenames specified will be used as the membernames for their respective entries in the TXTLIB file instead of the first CSECT in the file's text deck.
DEL	deletes members from a TXTLIB on a read/write disk and compresses the library itself to remove unused space. If more than one member exists with the same name, only the first entry is deleted.

Note: Unlike as in the MACLIB, there is no separate command to compress the library.

MAP lists the names (entry points) of TXTLIB members, their location in the library, and the size of each entry.

The DISK, PRINT, and TERM options of the MAP function operate the same way as for the **MACLIB** command:

DISK (the default) writes the listing to the A-disk with the filename corresponding to the name of the library and a filetype of MAP.

PRINT writes the listing to the A-disk with the filename corresponding to the name of the library and a filetype of MAP. It then prints the map on the spooled virtual printer.

TERM displays the TXTLIB map on the terminal.

All three options cause any existing MAP of the same name to be erased, but only the DISK and PRINT options create a new map.

Notes:

1. *You may add linkage editor control statements such as NAME, ALIAS, ENTRY, and SETSSI to a TEXT file before adding it to a TXTLIB. You must follow linkage editor conventions concerning format (column 1 must be blank) and placement within the TEXT file. The specified entry point must be located within the CSECT. See "Chapter 4: More about Compiling and Running a Program" on page 105 for a more complete discussion of the link-editing process and these statements.*

2. *The FILename option overrides any name card found in a text file. The name card functions as before, but the specified file name becomes the membername in the TXTLIB. The name card is the only entry point within that membername of the TXTLIB. If a name card is not found in the text file and you specify the FILename option, the file's name is the membername. The first CSECT in the text file is the first entry point (the remaining entry points in the text file follow) within that member.*

Loading an Object Module

Compiler output consists of relocatable object modules with a file type of TEXT. The code in these modules reference addresses relative to the start of an entry point. The reference point is always taken as zero.

When you want to run this kind of module, load it into storage at an address other than zero. In CMS, there are two main **user areas** of storage in which your programs execute:

user transient area	This is located starting at X'E000.' The transient area is fairly small (8K).
main user area	This are is used for larger programs. It begins at X'20000' in CMS storage and extends upwards to an area in high storage called **Loader Tables**. Generally speaking, the user area is large enough to hold very large programs or multiple programs constituting an application system.

To load an object module into storage, use the **LOAD** command. This loads the TEXT file into storage beginning at X'20000' unless otherwise specified. As the program is loaded, all address references within the module are resolved relative to the load point. Thus, if an object module references an address at X'30A' in the relocatable (TEXT) version, after issuing the **LOAD** command, all references to that address are changed to X'2030A'.

Once the program has been loaded into storage this way, it begins execution if you issue the **LOAD** command with the **START** option or issue the **START** command itself.

This directs CMS to pass control to your program, which continues until execution is completed. When finished, control returns to CMS. The **LOAD** command operates either on TEXT files or on individual members of TXTLIBs.

The **TXTLIB** command reads the object files as it writes them into the library. It creates a directory entry for each entry point or CSECT name or filename if the filename option is specified. Issue a **GLOBAL** command to define the library for the loader program, and specify the member name (the entry point) in the **LOAD** command.

Suppose you have an object module named TESTPRO1 TEXT that was added to a TXTLIB TESTLIB with the FILename option. At run time, you can issue the following command sequence:

```
global txtlib testlib
load testpro1 (start
```

Since the FILename option was specified in this example, the entry point and membername is TESTPRO1. So, the **LOAD** command specifies this entry point.

Suppose the FILename option was not specified and you have an object module named TESTPR01 TEXT that contains an entry point named TESTDATE. At run time you can issue the following command sequence:

```
global txtlib testlib
load testdate (start
```

In the latter example, the **GLOBAL** command defines the TXTLIB called TESTLIB TXTLIB. Your object module having the file identifier TESTPR01 TEXT is added to it. Since the entry point is TESTDATE instead of TESTPR01, the member name is TESTDATE. The **LOAD** command specifies this entry point. The option **start** passes control to that program once all address references are resolved by the loader.

If you want your TXTLIBs to be searched for missing subroutines during CMS loading processing, issue the **GLOBAL** command to identify the TXTLIB (just as you would for macro libraries):

```
global txtlib testlib
```

The **LOAD** command recognizes one entry point at a time. If more than one entry point is referenced, the **INCLUDE** command is used to reference additional entry points.

For example, suppose a program called PROG024 issues a CALL to another program called SUBCHECK. In this case you'd issue the command sequence:

```
load prog024
include subcheck
```

External references in PROG024 are resolved and the SUBCHECK module is loaded into storage. See "Loading Object Modules" on page 113 for further discussion of the loader module.

If the entry points exist in different TXTLIBs, the **GLOBAL** command must specify all the libraries that are required to resolve external references.

The GENMOD Command

When you have debugged and tested your programs, use the **GLOBAL**, **LOAD**, and **INCLUDE** commands together with the **GENMOD** command to create nonrelocatable object modules. These are executable modules whose external references have been resolved.

We'll continue our example from the previous section. The program PROG024 calls one subroutine module called SUBCHECK and another called TESTDATE. PROG024 and SUBCHECK exist as relocatable modules on your A-disk. They have filetypes of TEXT. TESTDATE is in a TXTLIB called TESTLIB. The command sequence:

```
global txtlib testlib
load prog024
include subcheck
include testdate
genmod prg24
```

This command sequence does the following:

- Brings all three modules into storage.

- Resolves all external references and addresses among them.
- Creates a nonrelocatable module called PRG24 MODULE A1.

The new module created by the **GENMOD** command can be executed directly in the CMS environment without having first to load it into storage. In our example, you can run the program simply by entering the following on the command line:

```
prg24
```

If PRG24 requires input and/or output files, you may have to define these files (using the **FILEDEF** command) before PRG24 can execute properly. If PRG24 expects arguments to be passed to it as parameters during execution, you can enter them on the command line following the MODULE name:

```
prg24 02/23/85
```

The **GENMOD** command always produces a file with a filetype of MODULE. It will replace any existing module of the same name.

The CMS Loadlib

LOADLIB is another type of library available to you. LOADLIBs, like MACLIBs and TXTLIBs, are in CMS simulated partitioned data set format. The members of LOADLIBs are link-edited programs that make use of certain OS macros such as LINK, LOAD, ATTACH, and XCTL. These macros require special handling by CMS at execution time, which is provided by the **OSRUN** command.

You create and manipulate LOADLIBs differently than you would MACLIBs and TXTLIBs. Use the **LKED** command to create a LOADLIB or LOADLIB member. Use the **LOADLIB** command to manipulate load libraries. This functions in a way similar to the **MACLIB** and **TXTLIB** commands.

The LKED Command

Use the **LKED** command to create a CMS LOADLIB or add members to an existing library. For example:

```
lked prog025 (list term disk
```

Use the **XREF**, **MAP**, and **LIST** options to cause the linkage editor to produce different types of documentary output.

XREF	produces an external symbol cross-reference for the modules being processed.
MAP	produces only a module map for the processed module.

LIST	(the default) includes only linkage editor control messages in the printed output file.

The **TERM** and **NOTERM** options cause the linkage editor to display diagnostic messages (the default) or to suppress such messages at the terminal.

Use the **PRINT, DISK**, and **NOPRINT** options to direct the linkage editor printed output to specific medium.

PRINT	spools the linkage editor printed output to the printer.
DISK	(the default) stores the linkage editor output in a CMS disk files with a filetype of LKEDIT.
NOPRINT	suppresses all printed output.

You can use other options with the linkage editor to specify characteristics of the load module:

LET	the module is marked as executable, even in the event of some linkage editor error condition.
NE	the module is noneditable. This means that it cannot be processed again by the linkage editor.
OL	the module is only loadable. The module cannot be accessed via any command except an OS LOAD.
RENT	the module is reentrant. The same copy of the module can be used concurrently by two or more tasks.
REUS	the module is reusable. The same copy of the routine can be used by two or more tasks (but not concurrently).
REFR	the module is refreshable. The module cannot be modified by itself or by any other module during execution.
OVLY	the module contains an overlay structure.

The linkage editor produces two permanent files on your A-disk (unless you specify PRINT or NOPRINT, in which case only one file is produced). The filename of both files is the name specified in the **LKED** command. The printed output of the linkage editor is given the filetype LKEDIT. The other file contains the load module(s) created by the linkage editor. It's given the filetype LOADLIB.

The OSRUN Command

With a knowledge of the linkage editor and its control statements, you can manipulate LOADLIBs to provide an organized library of executable OS modules. To make a load library ready for the **OSRUN** command, use the **GLOBAL** command, as with MACLIBs and TXTLIBs. For example, to execute a module called OSTEST1, use the command sequence

```
global loadlib oststlib
osrun ostest1
```

The **GLOBAL** command specifies the library to be searched. The **OSRUN** command performs the search, loads and relocates the member, and executes it. The **OSRUN** command searches only the libraries specified in the LOADLIB global list, unless you have a system library named $SYSLIB LOADLIB. In this case, OSRUN will search if it can't find the member name specified on the command line.

The LOADLIB Command

The **LOADLIB** command is a utility to maintain CMS LOADLIBs. Use this command to list the members of a LOADLIB, copy members from one LOADLIB to another, merge complete LOADLIBs, or compress a LOADLIB.

ISPF/PDF Libraries

Application programmers often work in groups to develop application programs. In many cases, a programmer is a specialist in certain areas of application programming, such as writing Assembler Language subroutines to be called by programs in high-level languages. You may be responsible for creating certain file structures for use in multiple applications.

For example, you may be asked to create and maintain the Data Division statements that define certain file structures to be used by a number of COBOL programs, written by other programmers. Or you may need to write certain FORTRAN subprograms to be called by main programs during the course of processing.

To help share source and object code, the **Interactive System Productivity Facility (ISPF)** has a companion product called **Program Development Facility (PDF)** that you can use to create and maintain libraries of shared source code, object code, data or documentation. These libraries may be sets of CMS files, MACLIBs, or TXTLIBs. They're identified by project name, group name, and type of information in the library.

An ISPF/PDF library is a collection of code or data units, called members. Each library generally contains members with the same type of information. For example, all the members of one library may consist of Assembler source code. Another could contain COBOL Data Division definitions, or documentation files written in SCRIPT. ISPF/PDF libraries are maintained internally as CMS files. Each library may consist of a set of CMS sequential files, or it may be a MACLIB or (for TEXT libraries only) TXTLIB. The particular organization is designated when the library is specified to PDF via the file utility (option 3.2).

Each ISPF/PDF library is identified by the following:

Project name	is the common identifier for all libraries belonging to the same project.
Group name	is the identifier for a particular set of libraries.
Type	is the identifier for the type of information in the library.

These characteristics are usually represented by PDF the same way an OS partitioned data set is represented: you'd join them with a period. For example, if your project name is PERSONNEL, the group name is TESTLIB, and the information type is COBOL, the library would be specified as:

```
PERSONNEL.TESTLIB.COBOL
```

Most projects use a hierarchy of related libraries to maintain effective version control over the programming development process and to reduce contention in library usage. For example, there may be three levels of library for a given project: a master library for production, a test library, and multiple development libraries. The master library designator could be PRODLIB, the test library TESTLIB, and the development library DEVLIB. The development library could also be given the name of the CMS user who owns the particular library.

For the PERSONNEL project, you could have the following library names:

PERSONNEL.PRODLIB.COBOL

PERSONNEL.TESTLIB.COBOL

PERSONNEL.DEVLIB.COBOL

Each library is uniquely named. This gives great flexibility in accessing various members contained in them.

Specifying ISPF/PDF Libraries and Their Members

To specify a member of an ISPF/PDF library, you must enter a project name, group name, type qualifier, and member name. Each of these items may contain up to eight alphanumeric characters. For the project name, group name, and type name, the first character must be alphabetic; for a member name, the name must follow CMS filename naming conventions. PDF automatically issues the appropriate **LINK** and **ACCESS** commands necessary to access the minidisk on which the library resides.

PDF panels prompt you for each component of the library identification as follows:

```
ISPF LIBRARY:
    PROJECT ===>
    GROUP   ===>
    TYPE    ===>
    MEMBER  ===>
```

To gain access to a member called TESTPROG, residing in the PERSONNEL.DEVLIB.COBOL library, for example, you would respond to the PDF panel prompts as follows:

```
ISPF LIBRARY:
    PROJECT ===> personnel
    GROUP   ===> devlib
    TYPE    ===> cobol
    MEMBER  ===> testprog
```

If you don't specify the member name, PDF displays a list of members of the library, which you can browse before selecting a specific member. Member lists are provided for PDF functions such as BROWSE (to examine a file), EDIT (to make changes to a file), MOVE, COPY, and so on.

Guidelines for Library Specifications

You must specify each ISPF library with the ISPF/PDF file utility (option 3.2) before it can be used. The name of the library along with the following information must be specified:

ISPF/PDF Library Attributes
organization, record format, and record length.

ISPF/PDF LIBRARY Location
owner's id and device address.

Link Access Mode
for update (write and multi-write, among others). See *ISPF/PDF for VM/SP Program Reference.*

Filetype for organization S (set of files).

Filename for organization M or T (MACLIB or TXTLIB)

An ISPF/PDF library takes one of three forms:

S is a set of CMS sequential files, all with the same filetype. The CMS filenames are the same as the ISPF/PDF library member names. The CMS filetype can be anything that uniquely identifies the set of files on a minidisk, such as COBOL, DATA, or TEXT.

M	is a CMS MACLIB, with a filename that uniquely identifies the MACLIB on the disk. The member names in the MACLIB are the same as the ISPF library member names.
T	is a CMS TXTLIB, with a filename that uniquely identifies the TXTLIB on the disk. The member names in the TXTLIB are the same as the ISPF library member names.

ISPF/PDF Library Record Format and Length

Libraries with an organization of **M** or **T** must have a record format of **F** (for fixed-length records) and a record length of 80. Libraries with **S** organizations may have **F** or **V** (variable length) formats, with record lengths from 1 to 32,767 bytes. (However, the PDF editor can only process records that are longer than 9 bytes and shorter than 256.)

Location of ISPF/PDF Libraries

Each ISPF library must be completely contained on one minidisk. You specify this with the userid of the owner, and the virtual address of the device on which the library resides.

You can have more than one ISPF/PDF library on the same minidisk. ISPF/PDF libraries can also exist on the same minidisk with other CMS files that aren't ISPF/PDF libraries. Usually, the lowest level libraries in a project (the DEVLIBs in our example) are private libraries, owned by the principal or only user. These should have an organization of **S** to eliminate the need for compressions. Higher level libraries are usually common libraries accessed for reading by anyone on the project, but maintained by one designated individual.

For example, if your responsibility is to maintain test data for a given project, you would have **write** access to the PERSONNEL.TESTLIB.DATA library. Everyone else on the project would only have **read** access. This kind of restriction helps protect the integrity of the data. It helps ensure that everyone is using the same files.

If you want to protect higher level libraries against unauthorized access by those outside the project, minidisks on which they reside can be protected with **read** passwords. You can, for example, assign the same **read** password to all minidisks containing libraries for the PERSONNEL project. This lets people working on the project to access any library, but prevents those outside the project from gaining access.

Concatenating ISPF/PDF Libraries

PDF lets you specify up to four libraries during source editing, compilation, assembly, or SCRIPT/VS processing (plus additional MACLIBs for compilations and assemblies). Generally, the lowest level library is specified ahead of the next higher level library, and so on, in bottom-to-top order. The following example shows how you could specify three libraries using the PDF library (member) specification panel:

```
ISPF LIBRARY:
    PROJECT ===> personnel
    LIBRARY ===> devlib    ===> testlib   ===> prodlib   ===>
    TYPE    ===> cobol
    MEMBER  ===> testprog
```

In this example, three libraries are specified in this order for TESTPROG COBOL:

PERSONNEL.DEVLIB.COBOL

PERSONNEL.TESTLIB.COBOL

PERSONNEL.PRODLIB.COBOL

Specifying libraries this way during editing lets you copy members to your development library. Use the specification sequence to search the libraries for the member you want to edit. The edited member is saved in your development library (the first library in the concatenation sequence), while the unchanged version remains in the test or master library. When you have finished testing the new version, you can promote it to a higher level library using the **move/copy** utility, PDF option 3.3.

Library concatenation during language processing makes it easy to include source segments via INCLUDE or COPY statements (or SCRIPT **imbed** controls). You can debug new or modified programs without altering the contents of the test or master libraries. The output from a compilation or assembly (object module) is stored in the lowest level TEXT library (the first library in the concatenation sequence).

ISPF/PDF Library Statistics

When a list of library members is displayed (for example, when you leave the MEMBER field blank on the PDF library selection panel), various statistics associated with each member are displayed, including:

Name of the member

Version number

Modification level

Creation date

Date last modified

Size.

These statistics help you keep track of files. Next to the name of the library member there's a blank field that you can use to SELECT a member for editing, browsing, or other PDF functions. You do this by placing the letter **S** in the blank letter field.

See *ISPF/PDF for VM/SP Program Reference* for additional information on ISPF/PDF libraries and the PDF functions.

Summary

Most operating systems provide **library** facilities. These help you develop programs and maintain an orderly environment for managing your files. A library is a CMS file that groups files (known as **members**) of a similar nature and function. To manipulate libraries and their members, you can use these library facilities, which are operating system functions.

All the CMS library types have a similar structure. Each one contains one or more members and has an **internal directory**. The library facilities use this directory to locate members. Since libraries are unlike other CMS files, you create, update, and use them differently than you do other CMS files.

There are three types of library facilities available in CMS. Load libraries are discussed in "The LOADLIB Command" on page 135. In this chapter we discussed the structure of macro and text libraries:

Macro are files (of filetype MACLIB) that contain one or more macros written in assembler language, or copy files written in other languages. By using the **MACLIB** command, you can create or change the contents of MACLIBs.

Text or program
are files (of filetype TXTLIB) created and altered by the **TXTLIB** command. They are libraries of code already compiled or assembled for use during program execution.

Chapter 6: Using Dialog Managers

Up to now, we've managed communication with the terminal simply by writing one line at a time, and reading one data item at a time. In many applications, this is all you need. Applications using several data items, however, are greatly simplified using **dialogs** between the user and the computer.

A common way to create dialogs is using full-screen displays, or **panels**. Although it's possible to create panels as data areas in programs, and write them to the terminal one line at a time, this uses a lot of storage and is time-consuming. Also, the task of dialog management itself — that is, controlling the flow from one panel to the next — can be very complex.

That's why it's better to use a standard data communications interface, or **dialog management system**. Two such systems are the **Interactive System Productivity Facility (ISPF)** and **Display Management System for CMS (DMS/CMS)**.

Using ISPF for Dialogs

The **Interactive System Productivity Facility (ISPF)** is an extension to VM/SP. It provides services that complement standard VM services, and that are designed just to implement interactive processing.

ISPF provides services to interactive applications that run under its control. As an application developer, you can use ISPF to:

- Display messages or predefined full-screen images (panels)
- Originate and maintain tables of user information
- Generate output files to be processed by other applications
- Define and control symbolic variables
- Control the various kinds of operational modes during processing
- Interface to Edit and Browse facilities (via ISPF/PDF).

Using Dialog Managers

An application that runs under ISPF is called a **dialog**. You can code your dialog in COBOL or FORTRAN. (For FORTRAN, you must use the VS FORTRAN Version 2, the VS FORTRAN, or the FORTRAN IV G1 compiler). You can even use more than one language in a dialog. There are also facilities that let you use the System Product Interpreter.

Each dialog is made up of various programs and data elements. There are five types of dialog elements, some of which are optional. These are the three most commonly used elements:

1. **Functions** are command procedures or programs that perform processing requested by you, such as display of panels and messages, building of tables, generation of output files, and control of operational modes.

2. **Panels** are predefined display images, such as menus, data entry panels, and information-only panels.

3. **Messages** are comments that provide special information to you, such as confirmation that a user-requested action is in process or completed, or a report of an error in the user's input.

There are two elements that aren't as commonly used:

1. **Tables** are two-dimensional arrays used to maintain data. Tables can be temporary or retained across sessions and shared among several applications.

2. **File Tailoring Skeletons** are generalized images of sequential data that can be customized during a dialog to produce an output file.

Panel definitions, message definitions, and skeletons are stored in libraries prior to execution of the dialog. You create them by editing directly into the panel, message, or skeleton libraries. No compiling or preprocessing step is required. Tables are generated and updated during dialog execution. Functions are programs or sequences of commands that you write to invoke and control the various ISPF elements and services.

In the following sections, we'll show you how to use ISPF to develop a dialog. In "Complete COBOL Program Using ISPF" on page 275, we give a complete COBOL program using ISPF. In "Complete FORTRAN Program Using ISPF" on page 281, we give a complete FORTRAN program using ISPF.

Developing an ISPF Dialog

To develop a dialog, you use an editor to enter the various components. You can use either the System Product Editor, or the edit option of the ISPF/Program Development Facility (ISPF/PDF).

You create panels by editing a file panel, defining the panel by means of keywords and options, and then saving the file as a member of an ISPF library.

A **panel definition** closely resembles the 3270 screen image that appears when the panel is displayed. Each character position in the panel definition is mapped to the same relative position on the display screen. You control where variable and literal data will appear by entering the variable name or literal itself on the panel definition exactly where you want it to appear.

You create messages in the same way, but they're saved in a **message library**. Each member of a message library can contain several messages, each one referenced by a unique **message id**. You specify the message text, the name of the corresponding HELP panel (to be displayed if the user requests help when the message is displayed), and an indication whether an audible alarm will be sounded. You can also specify a **short message** text to be displayed in the upper right-hand corner of the screen or some other position you specify.

You also create functions with the editor. Your COBOL or FORTRAN program can invoke ISPF services by calling an ISPF service interface routine called ISPLINK (in COBOL programs) or ISPLNK (in FORTRAN programs). On the call statement, you describe the services required. For example, suppose you have a panel called USRNAME in your panel library. To display USRNAME from a COBOL program you code:

```
WORKING STORAGE SECTION.
     77  DISPSERV       PICTURE A(8) VALUE 'DISPLAY '.
     77  PANEL          PICTURE A(8) VALUE 'USRNAME '.
     .
     .
     .
PROCEDURE DIVISION
CALL 'ISPLINK' USING DISPSERV PANEL
```

From a FORTRAN program, you invoke the same service like this:

```
INTEGER DSPSRV(2),PANEL(2)
DATA    DSPSRV/'DISP','LAY '/
DATA    PANEL/'USRN','AME '/
     .
     .
     .
LASTRC=ISPLNK(DSPSRV,PANEL)
```

The same panel can be displayed from an EXEC by invoking the **ISPEXEC** command:

```
ISPEXEC DISPLAY USRNAME
```

The **ISPEXEC** command lets you develop prototypes of functions using the System Product Interpreter. You can develop and test a dialog from an EXEC without writing a COBOL or FORTRAN application program. After you are satisfied with the dialog, you can simply translate your EXEC program into application language.

How to Begin Using ISPF

To use ISPF, certain requirements must be met. First of all, ISPF must be available to you, usually by means of a CMS system disk such as the S-disk or the Y-disk. If you are not sure where the ISPF program product resides, ask your supervisor. Each installation can install ISPF to suit their own needs, which can vary considerably. You will need the various libraries distributed with the ISPF program product.

The ISPF libraries distributed are:

ISPPLIB MACLIB Panel Libraries

ISPMLIB MACLIB Message Libraries

ISPSLIB MACLIB Skeleton Libraries

ISPTLIB MACLIB Table Input Libraries

You will also need the ISPSTART command to begin dialog processing. If these commands and libraries aren't available to you, consult your supervisor or system administrator.

Before you invoke ISPF, your virtual device 191 must be accessed as the A-disk. During operation, ISPF assumes that this minidisk is always in read/write mode and that no other user has write access to it. (In some cases, ISPF permits multiple write access to minidisks other than 191, provided that such access is performed under the control of ISPF.)

The libraries distributed with ISPF are system libraries. To make these as well as your own libraries available to applications running under ISPF control, you need to issue some **FILEDEF** commands, which should remain in effect throughout your ISPF session. Suppose you have a panel library called USRPANEL, and a message library called USRMESGS. You need to concatenate these libraries with the corresponding distributed libraries, and you want your libraries accessed ahead of the distributed libraries. The next sequence of commands (which can be included in your PROFILE EXEC or in another EXEC) make these libraries available to ISPF functions:

```
FILEDEF ISPPLIB DISK USRPANEL MACLIB * (PERM CONCAT)
FILEDEF ISPPLIB DISK ISPPLIB  MACLIB * (PERM CONCAT)
FILEDEF ISPMLIB DISK USRMESGS MACLIB * (PERM CONCAT)
FILEDEF ISPMLIB DISK ISPMLIB  MACLIB * (PERM CONCAT)
```

Notice that the ddname in each pair of **FILEDEFs** is the same as the file name of the distributed ISPF library. Other ISPF libraries follow the same pattern:

ISPSLIB is the ddname for all skeleton libraries

ISPTLIB is the ddname for all the table input libraries.

There are three optional libraries that are user-defined:

Skeleton library	ddname ISPSLIB
Table Output library	ddname ISPTABL
File Tailoring Output library	ddname ISPFILE

The PERM option ensures that the **FILEDEF** remains in effect throughout an ISPF session. The CONCAT option concatenates two or more libraries under the same ddname. The order in which libraries are searched is the same as the order in which the **FILEDEFs** are issued. (You don't have to issue a **GLOBAL MACLIB** command before invoking ISPF.)

If the ISPF commands and libraries aren't on a system disk, but are available by means of the **LINK** command, you might want to write an EXEC to link the ISPF system disk and issue the **FILEDEFs** you need. If the ISPF system disk is on a minidisk with a virtual address of 591, owned by a user called ISPMAINT, with a read password of ALL (that is, not requiring a password to link), the following statements in Restructured Extended Executor language do this:

```
/* ACCESS ISPF SYSTEM */
CP LINK ISPMAINT 591 591 RR
ACCESS 591 Z/A
FILEDEF ISPPLIB DISK USRPANEL MACLIB * (PERM CONCAT)
FILEDEF ISPPLIB DISK ISPPLIB  MACLIB * (PERM CONCAT)
FILEDEF ISPMLIB DISK USRMESGS MACLIB * (PERM CONCAT)
FILEDEF ISPMLIB DISK ISPMLIB  MACLIB * (PERM CONCAT)
```

Note: See “Chapter 8: EXECs” on page 191 for a discussion of EXECs.

You can create panel and message libraries by using the System Product editor together with the MACLIB command. Create each panel with the editor first, then build the panel library with the MACLIB command.

Note: The panels and groups of messages must have a CMS filetype of COPY. When using the editor to create a panel and to specify a file type of COPY, be sure to enter the editor sub-command SERIAL OFF to prevent the editor from inserting serial numbers in the panel file in columns 73 - 80. If these numbers are present, they will cause ISPF errors. You can also use a different filetype (for example, PANEL or MSG) and then rename the file before building the library.

The following steps outline a method of building a panel or message library:

1. XEDIT MENUPAN PANEL
2. (Create Panel)
3. FILE MENUPAN COPY
4. XEDIT NAMEPAN PANEL

5. (Create Panel)

6. FILE NAMEPAN COPY

7. MACLIB GEN USERPAN MENUPAN NAMEPAN

In steps 1 and 4, the panel members are created by using a filetype of PANEL to bypass serialization. In steps 2 and 5, edit sub-commands are used to create the panel members. In steps 3 and 6, a form of the FILE sub-command is used to write the files to disk with a filetype of COPY. In step 7, the MACLIB command is used to create USERPAN MACLIB. This library contains the two members MENUPAN COPY and NAMEPAN COPY.

After you create the panels and messages you need, you can develop a prototype application using the System Product Interpreter language to invoke the **ISPEXEC** command, or you can develop COBOL or FORTRAN applications.

Once your programs are compiled and exist either as text or load modules, you need to make them available to ISPF by issuing the appropriate **FILEDEF** command. For example, if you write a program called TESTPROG and compile it, you have a file called TESTPROG TEXT A1 on you A-disk. If you want to include TESTPROG in a TXTLIB called DEVLIB TXTLIB A1, issue the **TXTLIB GEN** or **TXTLIB ADD** command to insert the TEXT file into the library. This command makes the library available to ISPF:

```
FILEDEF ISPXLIB DISK DEVLIB TXTLIB * (PERM)
```

If you've included the module in a LOADLIB, use:

```
FILEDEF ISPLLIB DISK DEVLIB LOADLIB * (PERM)
```

When a text module is invoked (either as a TEXT file or as a member of a TXTLIB), any other text modules that it calls are loaded automatically by automatic call reference. The modules must also be TEXT files on a ISPF-accessible minidisk or members of the TXTLIB allocated to ddname ISPXLIB. If you have more than one TXTLIB, use the CONCAT option of the **FILEDEF** command to concatenate the libraries under the same ddname, ISPXLIB.

If your program is in a LOADLIB, use the ddname ISPLLIB. You can also specify a concatenated sequence for LOADLIBs. No automatic call referencing occurs with load modules. All load module references must be resolved prior to invocation by ISPF. Load modules can be used only for programs that are reenterable.

Note: Avoid using nonrelocatable (MODULE) files whenever possible. User MODULEs can create a very complex operational environment, since CMS subset mode is turned on to prevent MODULE files from overlaying relocatable programs already in storage. When using split screen mode, CMS subset mode isn't turned off until all relocatable programs associated with logical screens have completed execution.

When you've created the dialog functions you need, you can invoke the ISPF environment by means of the **ISPSTART** command, using the appropriate PANEL, CMD, or PGM parameter.

- The **PANEL** parameter causes the panel specified to be displayed, and passes any options to it that are specified on the **ISPSTART** command line.
- The **CMD** parameter specifies the name of an EXEC to be invoked as the first dialog function.
- **PGM** is used to specify the program name to be invoked as the first dialog function.

ISPF Dialog Organization

You can organize dialogs in a number of ways to suit the needs of the application. A typical dialog, for example, starts with a display of the highest menu in a hierarchy. This is the **primary option menu**. User options selected from this menu can invoke a dialog function, or display a lower level menu. The lower level menu can also cause functions to receive control, or pass control on to still other lower level menus. This hierarchical organization (tree structure) might look like Figure 11:

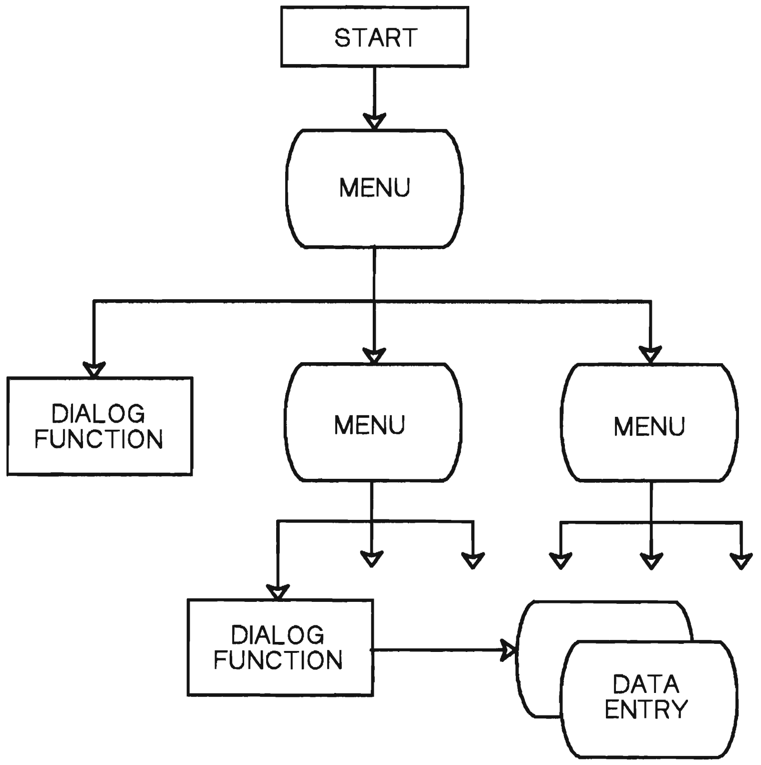


Figure 11. A Typical Dialog Starting with a Menu

Eventually, a dialog function receives control. When it does, it can use any of the dialog services provided by ISPF, including panel display for data entry. When the function completes execution, control is passed back up the tree to the panel from which the function was selected. Control eventually returns to the primary option menu. The process can now begin again with a different dialog path.

Controlling Dialog Flow with the SELECT Service

Your first major task in developing a dialog application is to design the dialog itself. That is, you have to define the structure and flow of panels, services and functions that make it up. Controlling the flow in a dialog is made possible by the SELECT service. The SELECT service is used by ISPF itself during its initialization to invoke a function or selection panel that begins a dialog. During dialog processing, SELECT can be used to display menus and invoke program or command procedure functions.

The same parameters used on the ISPSTART command line (PANEL, CMD, and PGM) can be passed to the SELECT service to specify the next action to be taken. If the CMD parameter is used, the EXEC it invokes can in turn invoke other EXECs, without requiring use of the SELECT service. When the PGM parameter is specified, the function it invokes can call other programs, which are considered part of the same function. If you call a function from within a program, use the SELECT service. Figure 12 on page 149 illustrates how the SELECT service is used to invoke and process a dialog.

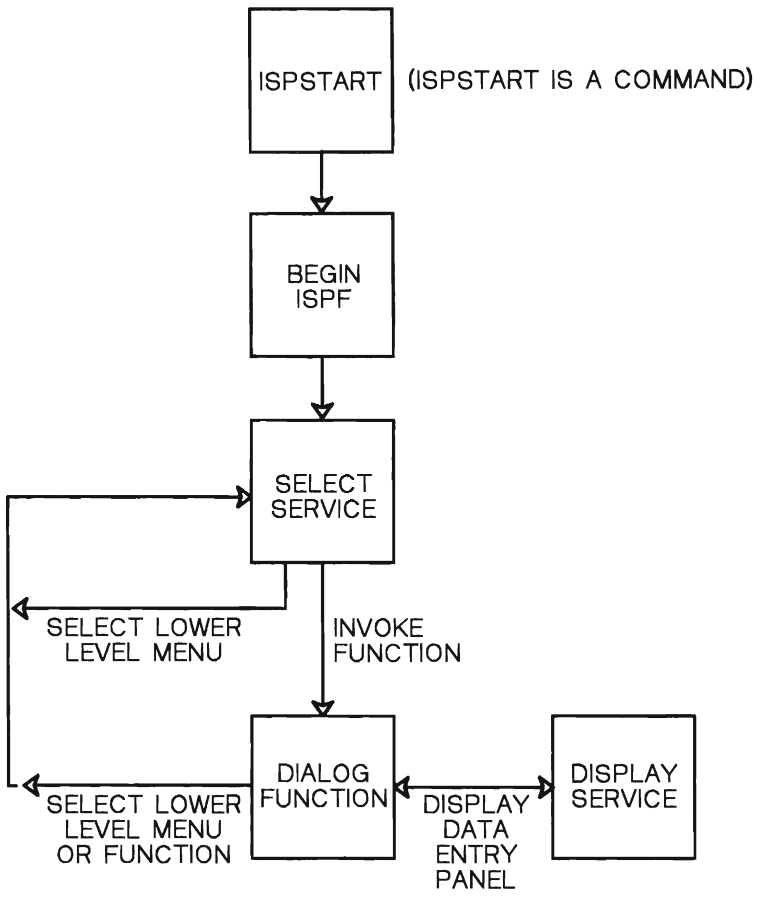


Figure 12. SELECT Service Used to Invoke and Process a Dialog

ISPF Panel Definition

You define a panel in ISPF using up to seven sections, of which only two (the BODY and END sections) are required for all panels. The PROC section is required for all selection panels. The seven sections are:

1. **Attribute section** defines the special characters used in the body of the panel definition to represent attribute (start-of-field) bytes, such as high intensity, low intensity, and input field.

2. **Body section** defines the format of the panel as seen by the user, and defines the name of each variable field on the panel.

3. **Initialization section** specifies the processing that will occur before the panel is displayed. You usually use this section to define how variables are to be initialized.

4. **Reinitialization section** specifies the processing that will occur prior to redisplay of a panel.

5. **Processing section** specifies the processing that will occur after the panel is displayed. You usually use this section to define how variables are verified and translated.

6. **Model section** (required for table display only; not allowed for other types of panels) specifies the format for displaying each row of the table.

7. **End section** consists of only the)END statement. ISPF ignores any data that appears on lines following the)END statement.

The panel display service recognizes these default field attribute characters:

+	text (protected) field, low intensity
%	text (protected) field, high intensity
	input (unprotected) field, high intensity.

Each panel definition section begins with a statement that indicates the section being defined. There are seven statements, one for the start of each of the sections. The statements are:

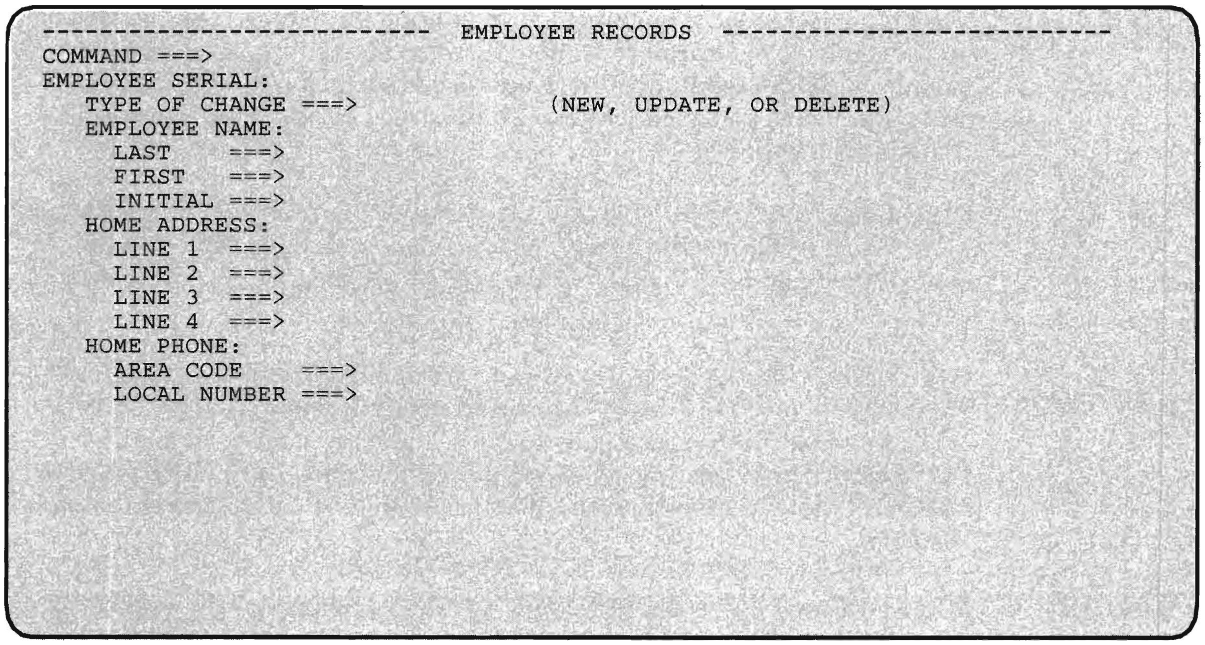
)ATTR	attribute section
)BODY	body section
)INIT	initialization section
)REINIT	reinitialization section
)PROC	processing session
)MODEL	model section (table displays only)
)END	end of panel definition

You can define all data entry panels of a dialog using only the)BODY and)END statements and the default field attributes. The screen definition below does not contain the other statements.

```
)BODY
%---------------------------  EMPLOYEE RECORDS  ------------------------------
%COMMAND ===>_ZCMD
%
%EMPLOYEE SERIAL: &EMPSER
+
+    TYPE OF CHANGE%===>_TYPECHG  +  (NEW, UPDATE, OR DELETE)
+
+    EMPLOYEE NAME:
+      LAST   %===>_LNAME          +
+      FIRST  %===>_FNAME          +
+      INITIAL%===>_I+
+
+    HOME ADDRESS:
+      LINE 1 %===>_ADDR1                              +
+      LINE 2 %===>_ADDR2                              +
+      LINE 3 %===>_ADDR3                              +
+      LINE 4 %===>_ADDR4                              +
+
+    HOME PHONE:
+      AREA CODE   %===>_PHA+
+      LOCAL NUMBER%===>_PHNUM    +
+
)END
```

Figure 13. Sample ISPF Panel Definition

When this panel is displayed to the user, it looks like this:



```
---------------------------  EMPLOYEE RECORDS  ---------------------------
COMMAND ===>
EMPLOYEE SERIAL:
   TYPE OF CHANGE ===>             (NEW, UPDATE, OR DELETE)
   EMPLOYEE NAME:
     LAST    ===>
     FIRST   ===>
     INITIAL ===>
   HOME ADDRESS:
     LINE 1  ===>
     LINE 2  ===>
     LINE 3  ===>
     LINE 4  ===>
   HOME PHONE:
     AREA CODE    ===>
     LOCAL NUMBER ===>
```

Figure 14. Sample ISPF Panel, When Displayed

For detailed information on how to define panels, see *ISPF Dialog Management Services and Examples* and *ISPF/PDF for VM/SP Guide.*

ISPF Message Definition

You create message definitions using an editor, such as the System Product Editor. They're saved in a member of the message library. As with panel definitions, no compilation is required. Each message in the message library consists of two lines. The first line contains the **message id** (required), **short message text** (optional), **name of corresponding HELP panel** (optional), and **audible alarm indicator** (optional). The second line contains the **full message text**.

The following message definitions contain all the fields:

```
MSG001   'OPERATION COMPLETED'     .HELP=MSGOK01    .ALARM=YES
'THE OPERATION SPECIFIED HAS BEEN COMPLETED.'
MSG002   'INVALID OPERATION'       .HELP=MSGNG01    .ALARM=YES
'ENTER A NUMBER FROM 1 TO 5 IN THE SPACE PROVIDED.'
```

If you want this message to be issued during a dialog, you refer to the message by the identifier MSG001. The panel MSGOK01 can be invoked by the user with the HELP service. When the message is displayed, the audible alarm sounds. Finally, a short form of the message is provided for display in the upper right hand corner of a panel, in case you don't want the full message displayed right away.

ISPF Variable Definition

Variable services let you define and use **dialog variables**. Dialog variables are the main communication vehicle between dialog functions (program modules or EXECs) and ISPF services. Program modules, EXECs, panels, messages, tables and skeletons can all reference the same data through the use of dialog variables.

The value of a dialog variable is a character string from zero to 32K bytes long. Some services restrict the length of dialog variable data; you can control the valid length of any dialog variable during panel and function definition.

You reference dialog variables by name. The name is composed of 1 to 8 characters (6 for FORTRAN). Alphanumeric characters (A-Z, 0-9, #, $, or @) can be used in the name, but the first character must be non-numeric. In the sample panel definition given above the names TYPECHG, LNAME, FNAME I, ADDR1, ADDR2, ADDR3, and ADDR4 are all names of dialog variables.

If you write a function in a language like FORTRAN or COBOL, identify the internal variables to be used as dialog variables to ISPF with the ISPF variable service VDEFINE. The program can also access and update dialog variables using VCOPY and VREPLACE. These services don't apply to EXECs.

ISPF Panel Services

You can use two ISPF panel services to manipulate panels.

DISPLAY is used to display data entry panels.

SELECT is used to display a hierarchy of selection panels (menus).

Use the DISPLAY service to control the display of individual panels, such as data entry, informational, or HELP panels. The SELECT service is used in a dialog to create a hierarchy of functions and menus that determine the sequence in which those functions and menus are processed.

The DISPLAY service reads a panel definition from the panel library, initializes variable panel fields from corresponding dialog variables, and displays the panel on the screen. A message can also be displayed with the panel.

The user can enter information in fields specified on the panel definition as input fields. After the user presses ENTER, the contents of the input fields are stored in dialog variables specified on the panel definition. Then, any processing specified on the panel definition using the)PROC statement is performed. The DISPLAY service returns to the calling function. Optionally, the cursor can be positioned at the start of any field in the panel definition.

If you want to invoke the DISPLAY service from your COBOL program, do so by defining the panel name, message-id, and field name in the Working Storage Section. For example, to display a panel called USRNAME, plus a message in the message library called PERX110, and to position the cursor at the field called LNAME, use this code:

```
WORKING-STORAGE SECTION.
     77 DISPSERV         PICTURE A(8) VALUE 'DISPLAY '.
     77 PANEL            PICTURE A(8) VALUE 'USRNAME '.
     77 PERX110          PICTURE A(8) VALUE 'PERX110 '.
     77 CURX110          PICTURE A(8) VALUE 'LNAME   '.
     .
     .
     .
PROCEDURE DIVISION.
     CALL 'ISPLINK' USING DISPSERV PANEL PERX110 CURX110.
```

From a FORTRAN program, the calling sequence is:

```
INTEGER DSPSRV(2),PANEL(2),PRX110(2),CRX110(2)
DATA    DSPSRV/'DISP','LAY '/
DATA    PANEL/'USRN','AME '/
DATA    PRX110/'PERX','110 '/
DATA    CRX110/'LNAM','E   '/
   .
   .
   .
LASTRC=ISPLNK(DSPSERV,PANEL,PRX110,CRX110)
```

From an EXEC, the command is:

```
   ISPEXEC DISPLAY PANEL(USRNAME) MSG(PRX110) CURSOR(CRX110)
```

You can also use the DISPLAY service to display messages, independently of panels. Do this by omitting the PANEL parameter; this causes the)REINIT section to be processed, and the current panel is overlaid with the message specified in the MSG parameter.

If you don't specify the panel-name or message-id, the)REINIT section is processed, and the current panel is redisplayed without a message.

You use the SELECT service to display and control a hierarchy of selection panels. Menus (selection panels) make up a special class of panels. A menu must have an input field to be used for the entry of selection options by the user of the application. This field, the standard name of which is ZCMD, is usually the first input field on line two of the panel. Corresponding to the ZCMD variable there must be a processing section in the panel definition in which ZCMD is translated and stored in the variable ZSEL. ZSEL is used by ISPF as input to the SELECT services. This parameter can be used to select a still lower panel definition. In this way, a path from the primary option menu can be defined down to the lowest level.

ISPF Variable Pools

To maintain multiple levels of control, dialog variables are organized into groups called **variable pools**, according to the dialog function and application with which they're associated.

A variable pool is basically a list of variable names that lets ISPF access the associated variables. When an ISPF service encounters a dialog variable name (in a panel, message table, etc.), it searches these pools to access the value of the dialog variable. There are three types of variable pool:

Function pool
contains variables only accessible by a given function.

Shared pool allows functions and selection panels to share access to dialog variables.

Shared pools are created by the SELECT service when it processes the **ISPSTART** or **ISPF** command and when the NEWAPPL or NEWPOOL keywords are specified with the SELECT service. When SELECT returns, the shared pool is deleted and the previous shared pool (if any) is reinstated.

Application profile pool
contains variables retained for the user from one ISPF session to another. Profile variables are automatically available when an application begins and are automatically saved when it ends.

ISPF Variable Services

A number of services are available in ISPF to control dialog variables:

VGET	retrieves variables from a shared pool.
VPUT	updates variables in a shared pool or profile pool.
VDEFINE	defines function variables.
VDELETE	removes definition of function variables.
VRESET	resets function variables.
VCOPY	copies data from a dialog variable to the program.
VREPLACE	copies data from the program to a dialog variable.

VGET and VPUT can be invoked from any function. The other variable services are for use from program modules only.

Like the panel and message services, you can invoke variable services from your COBOL program via the **ISPLINK** command, or from your FORTRAN program via **ISPLNK**. You can use the following COBOL statements to invoke the VDEFINE service. This defines the function variable LNAME, before displaying a panel containing the variable.

```
WORKING-STORAGE SECTION.
     01 VDEFINE                   PIC X(18)  VALUE "VDEFINE".
     01 NLNAME                    PIC X(7)   VALUE "(LNAME)".
     01 LNAME                     PIC X(16)  VALUE SPACES.
     01 CHAR                      PIC X(8)   VALUE "CHAR    ".
     01 LLNAME                    PIC 9(6)   VALUE 16 COMP.
     .
     .
     .
PROCEDURE DIVISION.
     CALL 'ISPLINK' USING VDEFINE NLNAME LNAME CHAR LLNAME
```

In this example, NLNAME is the name of the function variable — that is, LNAME. The level-01 name LNAME is the field to contain the value of the variable function. LNAME is initialized to spaces. CHAR is the literal **CHAR**, which indicates the format of the variable. LLNAME is the length of the variable field, 16 bytes.

In FORTRAN, these statements define the LNAME variable and initialize it:

```
IMPLICIT INTEGER (A-Z)
DIMENSION LNAME(4)
LASTRC = ISPLNK('VDEFINE', '(LNAME)',LNAME, 'CHAR',16)
```

Other ISPF Services

Other services are available in ISPF for dialog management. You can invoke each service from a program as shown for ISPLINK (COBOL) or ISPLNK (FORTRAN).

Table Services

ISPF **table services** let you maintain and access sets of dialog variables. A table is a 2-dimensional array of information in which each column corresponds to a dialog variable. Each row contains a set of values for those variables.

A table can be either temporary or permanent. Temporary tables exist only in virtual storage and can't be written to disk storage. Permanent tables are created in virtual storage, but can be saved on disks.

File Tailoring Services

Another type of ISPF service is the **file tailoring** service. These services read skeleton files from a library and write tailored output that can be used to drive other functions. The file tailoring output can be directed to a library and/or a sequential file that's been specified by the ISPF function. It can also be directed to a temporary sequential file provided by ISPF.

Each skeleton file is read record-by-record. Each record is scanned to find any dialog variable by name. When a variable name is found, its current value is substituted from a variable pool.

The file tailoring services are:

FTOPEN	prepares the file tailoring process. It specifies whether the temporary file will be used for output.
FTINCL	specifies the skeleton to be used, and starts the tailoring process.
FTCLOSE	ends the file tailoring process.
FTERASE	erases (deletes) an output file created by file tailoring.

Miscellaneous Services

In addition to display, variable, table and file tailoring services, ISPF provides **EDIT, BROWSE, LOG**, and **CONTROL** services.

The EDIT and BROWSE services are available only if PDF is installed. These services let you invoke the PDF edit or browse programs from a dialog function, specifying a CMS file.

The LOG service lets a dialog function write a message to the ISPF log file, which can be used as an audit or tracking mechanism.

The CONTROL service lets a dialog function condition ISPF to expect certain kinds of display output, or to control the disposition of errors encountered by ISPF services. The CONTROL service lets you:

- LOCK the terminal keyboard during a display
- Split a display screen if required (or inhibit screen splitting)
- REFRESH the entire screen on the next display
- Permit panels to be processed without displaying them.

Error-handling CONTROL parameters lets you terminate the dialog function upon receipt of a return code of 12 or higher (CANCEL parameter), or to RETURN control to the dialog function on all errors.

See *ISPF Dialog Management Services and Examples* for additional information on ISPF.

Using DMS/CMS for Dialogs

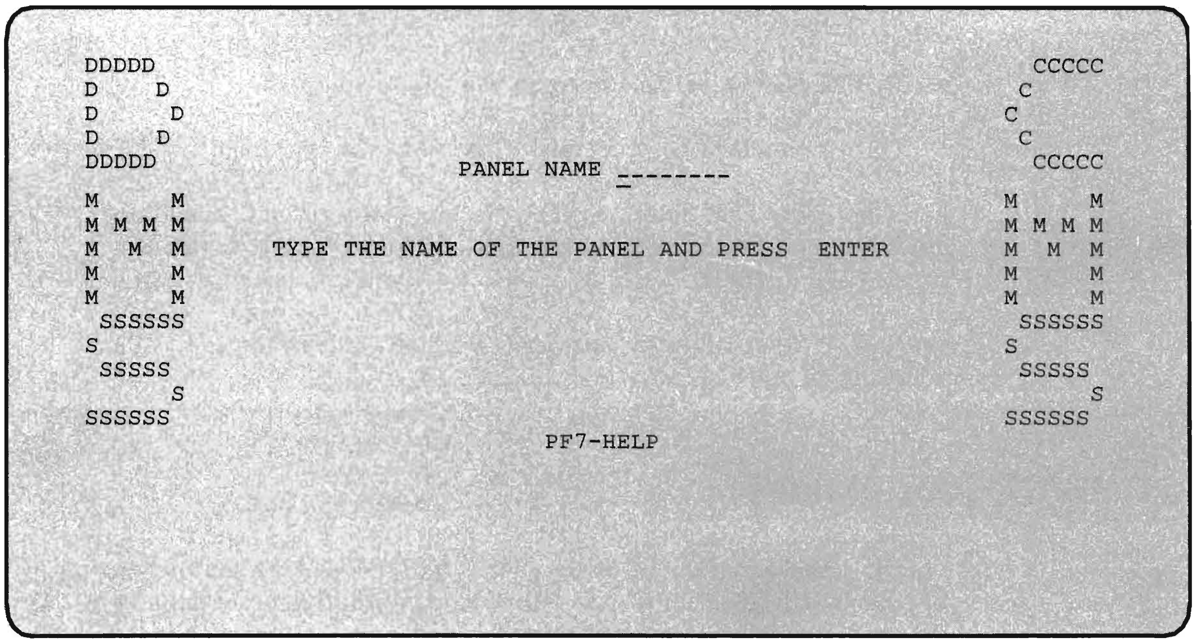
The Display Management System for CMS (DMS/CMS) is an extension to VM/SP that provides a way to implement interactive processing in VM. DMS/CMS lets you design full screen images (called **panels**) that can be displayed from applications written in Restructured Extended Executor (REXX) language, COBOL, or EXEC 2. Data entered into the various fields of a given panel are passed to the program using special interfaces available with EXEC 2.

DMS/CMS has three functional parts:

1. **Panel Formatter** used by panel designers to design the content and format of panels.
2. **Panel Manager** used by programmers to associate their REXX, COBOL, or EXEC 2 applications with defined panels.
3. **Write Full Screen** used by Assembler Language programmers to take advantage of the full screen I/O capabilities of DMS/CMS for 3270-type graphics devices. If you aren't coding in Assembler Language, you can ignore this.

Using the Panel Formatter

The Panel Formatter is the part of DMS/CMS you use to design panels. You invoke the Panel Formatter with the CMS command **PANEL.** When you do, this screen image will appear:



```
DDDDD                                                           CCCCC
D   D                                                          C
D    D                                                        C
D   D                                                          C
DDDDD                      PANEL NAME ________                  CCCCC
M     M                                                       M     M
M M M M                                                       M M M M
M  M  M       TYPE THE NAME OF THE PANEL AND PRESS  ENTER     M  M  M
M     M                                                       M     M
M     M                                                       M     M
 SSSSSS                                                        SSSSSS
S                                                             S
 SSSSS                                                         SSSSS
      S                                                             S
SSSSSS                                                        SSSSSS
                                  PF7-HELP
```

On this panel, you assign a name to the panel you'll be designing in the screens that follow. When you press the ENTER key, you move to the **Panel Size** screen. Here, you specify character width and the number of lines for the screen. When you press the ENTER key, you see the **Design Grid** screen. This is where you design the layout and content of your panel. The next screen is the **Field Definition** screen, where you define the characteristics of the fields you specified in your panel design. Finally, you go to the **Completion Options** screen, from which you store your panel for later use.

Here's a summary of the screens you use to design and store a panel:

Panel Name This screen is displayed when you enter the command **PANEL.** You can name your panel on this screen in the field indicated. The cursor is positioned at the start of this field when the panel is first displayed. When you press the ENTER key, you go the Panel Size screen.

Panel Size This screen is displayed after the Panel Name screen. You can also press the PF1 key at any time during a DMS/CMS session to review or modify the panel size definitions. You can specify the width of display lines (80 or 132 characters) and the number of lines on the screen. The default is 80 characters by 24 lines.

Design Grid This screen is displayed when you press PF2. You design your panel on this screen by entering and positioning text and data fields. This screen provides a grid giving row and column numbers to help you.

Field Definition
This screen is displayed when you press the PF3 key. You use this screen to define the characteristics of the fields you put in your panel on the Design Grid screen. The field characteristics you can define are:

- Intensity or color
- Extended highlighting
- Whether the field is protected (display only) or unprotected (able to receive data)
- Whether the field should be checked for alphanumeric or numeric values
- Whether the cursor should skip to the next field when the last character of the current field is reached.

Completion Options
This screen is displayed when you press the PF5 key. Use this screen to tell DMS/CMS whether to save your panel for later use and whether you want to design another panel at this time.

Designing Fields in DMS/CMS Panels

When you're preparing your field using the Design Grid screen, type a field wherever you want it displayed. You can then describe it to the panel formatter by preceding it with a character that identifies it as one of three types:

1. **Text Field** is preceded by the logical not character (¬) or by the exclamation point (!). Text fields are used as titles, identifiers, or instructions. Text fields preceded by the logical not character (¬) are displayed in normal intensity with no extended highlighting. Text fields preceded by the exclamation point (!) can have the default definitions changed, if you want, from the Field Definition Screen.

2. **Data Field** is preceded by the underscore character ( ). Data fields are used for the passage of data. These are the fields where the panel user will type information. Data fields may also be filled in with data supplied by a program or EXEC procedure when displayed to the panel user.

3. **Selector Field** is preceded by a percent sign (%). Selector fields are fields that are light-pen selectable. The panel user can touch a light-pen to the selector field to indicate a selection.

Each field you specify on the Design Grid screen must be preceded by one of the delimiters, which are specified on the bottom of the Design Grid screen. You can change any of these listed delimiters simply by typing the new delimiter over it. You'd do this, for example, if you were going to use one of the default delimiters (the exclamation point, for instance) in the text.

You indicate blanks or nulls in a field by using the appropriate characters. The at sign (@) is used to indicate blanks, and the number sign (#) is used to indicate nulls.

DMS/CMS provides a number of commands for your use only on the Design Grid screen. You type these commands on the command line (the bottom line) of the screen, right after the arrow (= = = >). To position the cursor at the command line, press PF6. Type the command you want and press ENTER or PF2 to execute the command. Here are the commands you can use:

DISPLAY	Display the screen as the user would see it.
ADD	Add blank lines to the screen.
DELETE	Delete lines from the screen.
DUPLICATE	Duplicate lines.
MOVE	Delete and move lines.
COPY	Duplicate and move lines.
LEFT	Move typed entries to the left.
RIGHT	Move typed entries to the right.
CENTER	Center entries on the screen.
TOP	Position starting with top line.
BOTTOM	Position starting with bottom line.
FORWARD	Move display toward bottom.
BACKWARD	Move display toward top.
NULLS	Blanks become nulls or vice versa.
CASE	Display as upper case or mixed case.

To find out more about these commands, see *Display Management System for CMS: Guide and Reference.*

When you've finished designing your panel layout using the Design Grid screen, you pass on to the Field Definition screen. One Field Definition screen contains the specifications for one field. The line that contains the field being considered is shown at the top of this screen, with the field itself intensified. On the next line, information is provided about the field: where it is (row and column), its length, what type it is, and its sequence in the panel.

Specify all fields on one line of the Design Grid screen before making any entries for any of them on the Field Definition screen.

There are no Field Definition screens for text fields preceded by a **logical not (¬)** character. This character indicates that the field is to be displayed at normal intensity without extended highlighting. For all other field types, you can specify intensity, color, and whether extended highlighting is to be used. This includes text fields preceded by an exclamation point (!). For data fields, you can also specify whether the field is to be protected or not. Unprotected fields can be used to accept data from the user. Protected fields won't accept any user data entry. The AUTOSKIP AT FIELD END attribute of data fields defaults to Y (yes). This causes the cursor to skip to the next data field when the end of the current field is reached during data entry.

Using the Panel Manager

The Panel Manager is the part of DMS/CMS you use to associate your COBOL, REXX, or EXEC 2 application with defined panels.

After you've designed and stored your panel, you can use it in your application program. To pass information to DMS/CMS from your COBOL program, you need to include this statement in the Working-Storage Section of your program:

```
    COPY EUDCOBOL
```

When you need to display a panel, you can code statements in the Procedure Division as in the following example:

```
ENTER LINKAGE.
CALL 'EUDCOBOL' USING EUDCNTRL D1 D2 D3 D4 D5 D6 FIELDS CASE
                      ALPHA-JUST ALPHA-FILL NUM-JUST NUM-FILL
                      RETURN-KEY RETURN-CURSOR
                      RETURN-CURSOR-OFFSET
                      DMASKS TMASKS SMASKS SFIELDS.
ENTER COBOL.
```

If you inserted the COPY EUDCOBOL statement in your Working-Storage Section, The **CONTROL** parameter will be EUDCNTRL. There's a listing of EUDCNTRL, which consists of a level-01 structure plus a number of level-77 data definitions, in *Display Management System for CMS: Guide and Reference.*

The **parameter list** consists of a number of directions to DMS/CMS for managing the panel specified in the control section. Using these parameters, you can specify these items:

load list	Areas in your program containing data to be loaded in the panel. The LOAD-LIST field in EUDCNTRL must be set to Y, and the number of areas specified in the load list must be equal to the value in the NUMBER-DATA-FIELDS field in EUDCNTRL.
unload list	Areas in your program where data from the panel is to be unloaded. The UNLOAD-LIST field of EUDCNTRL must be set to Y, and the number of areas specified in the load list must be equal to the value in the NUMBER-DATA-FIELDS in EUDCNTRL.
datamask	A structure in your program containing an entry for each data field that indicates how the field is to be displayed. The datamask field must be defined as: `PIC 999 COMP.` The EUDCNTRL section includes a number of level-77 values that you can use to set masks for color, intensity, highlighting, and protection. If a datamask parameter is to passed in the calling sequence, the DATA-MASK field of EUDCNTRL must be set to Y.
textmask	A structure in your program containing an entry for each text field indicating how the field is to be displayed. The textmask field must be defined as: `PIC 999 COMP.` The EUDCNTRL section includes a number of level-77 values that you can use to set masks for color, intensity, highlighting, and protection. If a textmask parameter is to passed in the calling sequence, the TEXT-MASK field of EUDCNTRL must be set to Y.
selectmask	A structure in your program containing an entry for each select field indicating how the field is to be displayed. The selectmask field must be defined as: `PIC 999 COMP.` The EUDCNTRL section includes a number of level-77 values that you can use to set masks for color, intensity, highlighting, and protection. If a selectmask parameter is to passed in the calling sequence, the SELECT-MASK field of EUDCNTRL must be set to Y.

select-items If you specify a selectmask and NUMBER-SELECT-FIELDS is greater than zero, you must pass a structure to DMS/CMS that contains one entry for each selector pen field.

If a panel is to be displayed, before issuing a call to EUDCOBOL, your program must move the panel name to the PANEL-NAME field in EUDCNTRL and set the DISPLAY-CODE to D.

The Panel Manager can hold control information for up to ten panels concurrently. If a display of more than ten panels is required of an application, your program can request that the Panel Manager release a currently active panel. A call to EUDCOBOL without a parameter list, specifying the panel to be released, does this. This is an example of how to release a panel named PERS001:

```
MOVE 'PERS001' TO PANEL-NAME.
MOVE 'P' TO DISPLAY-CODE.
ENTER LINKAGE.
CALL 'EUDCOBOL' USING EUDCNTRL.
ENTER COBOL.
```

The P in DISPLAY-CODE tells the Panel Manager to release the panel specified in PANEL-NAME. If PANEL-NAME is blank, the Panel Manager releases all currently active panels.

You can find more coding rules for COBOL in *Display Management System for CMS: Guide and Reference.*

Using EXECS to Prototype DMS/CMS Applications

You can use EXECs written in EXEC 2 or Restructured Extended Executor to prototype DMS/CMS applications to be written in COBOL. You do this using the EUDEXEC2 command.

To invoke DMS/CMS panel display services from an EXEC, you code DMS/CMS EXEC 2 commands in your EXEC and then invoke the EXEC by issuing EUDEXEC2. For example, if you code DMS/CMS EXEC 2 commands in an EXEC named TSTPROGX, use a command to invoke the EXEC, as in the following example:

```
EUDEXEC2 TESTPROGX EUDXPANL
```

The parameters you specify, if any, are passed to your TSTPROGX.

You can also invoke EUDEXEC2 from within an EXEC, using the EXEC 2 subcommand environment. To do this from TSTPROGY EXEC, for example, you code:

```
&TRACE
&STACK LIFO T1 T2 T3 T4 T5 T6 D1 D2 D3 D4 D5 D6 S1 S2 S3 S4 S5 S6
&READ STRING &ALL
EUDEXEC2
&PRESUME &SUBCOMMAND DISPLAY
MSGMODE OFF
USE PANEL EUDXPANL
    .
    .
    .
```

The DMS/CMS EXEC 2 commands are as follows:

USE	Indicates the panel being used.
DISPLAY	Causes the panel to be displayed.
MAP	Associates a name with a panel field.
SET	Dynamically changes field attributes.
RESET	Resets any changed attributes.
CURSOR	Positions the cursor on the displayed panel.
SIGNAL	Sounds alarm when the panel is displayed.
COMMENT	Places a comment on the bottom line.
CASE	Specifies upper or lower case of data entered by the user.
NUMBER	Determines the handling of numeric fields.
ALPHANUM	Determines the handling of alphanumeric fields.
MSGMODE	Suppresses DMS/CMS error messages.
TERMINATE	Indicates that panel processing is completed.

For detailed descriptions of these commands, see *Display Management System for CMS: Guide and Reference.*

Summary

In this chapter we've discussed a way to create dialogs using full-screen displays, or **panels**. The task of dialog management itself — that is, controlling the flow from one panel to the next — can be very complex. That's why it's better to use a standard data communications interface, or **dialog management system**. Two such systems discussed in this chapter are the **Interactive System Productivity Facility (ISPF)** and **Display Management System for CMS (DMS/CMS)**.

The **Interactive System Productivity Facility (ISPF)** is an extension to VM/SP. It provides services that complement standard VM services, and that are designed just to implement interactive processing.

ISPF provides services to interactive applications that run under its control. The Display Management System for CMS (DMS/CMS) is an extension to VM/SP that provides a way to implement interactive processing in VM. DMS/CMS lets you design full screen images (called **panels**) that can be displayed from applications written in REXX, EXEC 2, or COBOL. Data entered into the various fields of a given panel are passed to the program using special interfaces available with EXEC 2.

For the complete COBOL program example using ISPF, see "Complete COBOL Program Using ISPF" on page 275. For the complete FORTRAN program example using ISPF, see "Complete FORTRAN Program Using ISPF" on page 281. For examples of complete ISPF screen definitions, see Appendix C, "ISPF Panels" on page 283.

Chapter 7: Using SQL/DS

Another useful facility is the **data base**, a centrally controlled, integrated collection of data, along with a **Data Base Management System (DBMS)** that controls the storing and retrieval of data. Data base systems are useful because they can be used to:

- Reduce redundancy
- Avoid inconsistencies
- Share data among many users
- Enforce data processing standards
- Apply and maintain data integrity and security
- Resolve conflicting application requirements.

The **Structured Query Language/Data System (SQL/DS)**, a full scale data base management system completely integrated into the VM/SP environment, is available for CMS users.

Structured Query Language/Data System (SQL/DS) is a relational data base management system designed for end users. SQL/DS simplifies data handling by offering facilities for querying and manipulating data and writing reports. It also contains data recovery and data security measures.

This chapter is intended to provide a general introduction to the programming facilities of SQL/DS. It is not intended to be a complete description of the use of SQL/DS. For more complete information on SQL/DS, see *SQL/DS Application Programming.*

In the sections that follow, we'll discuss how to use SQL in FORTRAN and COBOL programs. For a more complete COBOL program and FORTRAN program using SQL/DS statements, see the appropriate appendixes in *SQL/DS Application Programming.*

In addition to using SQL/DS in programs, you can use it directly from your terminal via the ISQL facility of SQL/DS. This is particularly useful for one-time, set-up, and administrative functions. It is also useful for prototyping commands that you plan to use in your programs.

How SQL Handles Data

In SQL/DS, the data is addressed by content, rather than by its location or organization in storage. It takes the form of tables in row and column format. SQL/DS also keeps catalogs that serve as an integrated data dictionary and directory. These catalogs always reflect the current status of the data base and are automatically updated.

Data is defined and accessed in terms of tables and operations on tables. A table is defined to SQL/DS by identifying the **columns** in the table and their characteristics. All values in a column have the same characteristics. A table **row** is the smallest unit of insertion and deletion in SQL/DS. An insert operation adds one or more rows to a table. A delete operation removes one or more rows from a table. The smallest unit of data update in SQL/DS is the **field**, which is the point where a specific row and column meet. A field contains a single **data item**.

You can do the following table operations:

- Create or delete tables.
- Retrieve data by table, row, or field.
- Update, insert, or delete data.
- Add new columns to a table.
- Copy data from one table into another.
- Perform utility operations, such as bulk data loading, data reorganization, and printing.

SQL/DS can also store **indexes** to particular columns in a table. You don't need indexes to access stored data, but they help SQL/DS optimize its performance. When you request an index, SQL/DS creates and maintains it. When you write a program to access data, you don't refer to the index explicitly, but SQL/DS decides which index to use.

SQL/DS can also store **view definitions**. A view is a logical or virtual table derived from one or more tables. It's like a stored table with rows and columns. You can use views as if they were tables. However, some operations are not valid on views. Others are restricted, depending on how the view was defined. You can use views mainly to simplify data retrieval commands and to limit access to data or its manipulation.

Using SQL, you specify only the results you want. When you reference the data, you don't specify data paths, access methods, or the organization of the physical file.

SQL Commands

An SQL command contains a verb with one or more optional clauses, language keywords, and parameter operands. The structured use of verbs and keywords in the SQL syntax lets you request data in readable form.

The SQL commands most commonly used are the QUERY command, the DATA MANIPULATION commands, and the DATA DEFINITION commands. When used in a program, the QUERY command is used inside a DECLARE CURSOR command so that you can FETCH rows of the QUERY result individually.

The QUERY command is:

SELECT	Retrieves data from one or more tables.

The DATA MANIPULATION commands are:

INSERT	Places a new row in a table.
UPDATE	Changes field level data.
DELETE	Removes one or more rows from a table.

The DATA DEFINITION commands are:

CREATE TABLE	Defines a new table and its columns.
DROP TABLE	Erases a table.
ALTER TABLE	Adds new columns to a table.
CREATE INDEX	Defines an index that lets you access rows of a table in a specific sequence.
DROP INDEX	Erases an index.
CREATE VIEW	Defines a logical table from one or more tables or views.
DROP VIEW	Erases a view definition.

SQL/DS operates in three modes:

1. **Single User Mode** lets a single application or utility perform in the same virtual machine as SQL/DS. It is used primarily for development and testing. This mode is also intended for dedicated functions like bulk loading and unloading data bases, and other situations that may require a dedicated SQL/DS data base.

2. **Multiple User Mode** lets you and other users or operations access the same data base at the same time. It's the most common operational mode. The advantages are shared access and SQL/DS isolation from individual applications through isolation of virtual machines.

3. **Multiple Data Base Mode** Lets several SQL/DS data bases run at the same time in either multiple or single-user mode.

Using the SQL/DS Preprocessors

COBOL or FORTRAN programs can issue SQL commands by imbedding them in line with standard language statements. For example, in a COBOL program you can define data areas in Working-Storage to receive data accessed by SQL commands imbedded in the Procedure Division:

```
WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  EMPSER          PIC X(6) VALUE SPACES.
01  FNAME           PIC X(16) VALUE SPACES.
01  LNAME           PIC X(16) VALUE SPACES.
    EXEC SQL END DECLARE SECTION END-EXEC.
   .
   .
   .
PROCEDURE DIVISION.
    EXEC SQL DECLARE CRSR CURSOR FOR
    SELECT FRSTNAME, LASTNAME
    FROM NAMELIST
    WHERE SERIALNO = :EMPSER
    END-EXEC.
```

In FORTRAN, you code the following statements to imbed the SQL **SELECT** command:

```
 EXEC SQL BEGIN DECLARE SECTION
 CHARACTER*6 EMPSER
 CHARACTER*16 FNAME,LNAME
 EXEC SQL END DECLARE SECTION
.
.
.
 EXEC SQL DECLARE CRSR CURSOR FOR
*SELECT FRSTNAME, LASTNAME
*FROM NAMELIST
*WHERE SERIALNO = :EMPSER
```

Note: The asterisks (*) in the above example are required continuation characters.

In these examples you can access a table called NAMELIST in the SQL data base. FRSTNAME (first name) and LASTNAME (last name) are columns. Data is selected from these columns when an EMPSER defined in the program is matched with a serial number value in the table. SQL brings in the the data and inserts it into the fields defined in the program.

SQL/DS analyzes and converts the embedded SQL commands to SQL/DS calls before the normal COBOL or FORTRAN compilation. The keyword EXEC isn't defined for either the COBOL or FORTRAN compiler. SQL commands are converted to equivalent language statements before they're compiled. This is done using a SQL/DS **Preprocessor**.

Preprocessing does two things: First, it modifies the source program by replacing SQL statements with standard host language code. (SQL statements are kept as comments.) The source program is then ready for normal processing. Second, it optimizes and compiles the SQL statements by defining them to SQL/DS and creating an access module that efficiently executes the SQL requests that the program makes.

After the SQL/DS Preprocessor has processed your program, you can use the FORTRAN or COBOL compiler to produce an executable object module.

When you run a program, the access module created by the SQL/DS Preprocessor is called to handle each SQL command. The access module resides in the SQL/DS data base. At preprocessing, SQL/DS chooses the best access path for each SQL command in the program, based on available indices, data statistics, etc., and stores the access information in the access module. When SQL/DS loads the access module, it checks that the module is still valid. An access module may be invalid, for example, if a path is based on an index that is no longer available.

Declaring Host Variables to SQL

A **host variable** is a variable referenced by SQL in your program. SQL/DS recognizes two types of host variables: **main variables** and **indicator variables**.

Main Variables

Main variables are normal program variables used in SQL statements. To get SQL to recognize these variables, you must place them in a SQL Declare Section. This is a special area in your program that's delimited by two SQL statements:

- BEGIN DECLARE SECTION
- END DECLARE SECTION

Main variable names can have as many as 18 characters in COBOL and 6 in FORTRAN. This can consist of A-Z, 0-9, the three national characters (@, #, $), and the underscore. (You can also use the hyphen in COBOL, where the preprocessor treats it internally as an underscore.) In COBOL, don't give any variable a name beginning with SQL or RDI. In FORTRAN, don't give any variable a name beginning with SQ. These are reserved for SQL/DS use.

You can't use a number or the underscore as the first character of a variable name. Other naming restrictions apply to specific languages. When you reference variables in SQL statements, preface them with a colon. For example, a variable named DBDESC is referenced as :DBDESC in a **SELECT** command.

Indicator Variables

By using optional indicator variables, you can indicate null values on input to SQL/DS (the **UPDATE** and **INSERT** statements) or output from SQL/DS (the **INTO** clause of a FETCH statement). You must declare indicator variables in the SQL Declare Section. They must be of a host language data type equivalent to the SQL data type **SMALLINT**. In COBOL, this is S9(4) COMP; in FORTRAN it's INTEGER*2. When used in an SQL statement, the indicator variable names must follow the corresponding main variable name and must be preceded with a colon. For example, if the main variable name is DBDESC and the corresponding indicator name is DESCIND, in a SQL statement you'd refer to it with the expression :DBDESC:DESCIND.

After an SQL request involving an output variable is satisfied, a value is returned to your program in the indicator variable.

- When the indicator variable value is zero, the value returned into the main variable isn't null.
- When the indicator variable value is negative, the main variable is null and should not be used for processing by the host program.
- When the indicator variable value is positive, SQL/DS has truncated the value of the main variable. The returned value was larger than the declared value.

Data Types Supported by SQL/DS

SQL/DS supports the following data types in the corresponding COBOL or FORTRAN formats:

SQL Datatype	COBOL Equivalent	Remarks
INTEGER	PIC S9(9) COMP.	
SMALLINT	PIC S9(4) COMP.	
FLOAT	COMP-2.	
CHAR(n)	PIC X(n).	n is the number of characters.
DECIMAL(m,n)	PIC S9(p)ﬃV9(n)“ COMP-3.	m is the precision. n is the scale (the number of digits to the right of the decimal). p is the number of digits to the left of the decimal.
GRAPHIC(n)	PIC G(n) DISPLAY-1.	n is the number of graphic characters.
VARCHAR and LONG VARCHAR	01 S-VAR. 49 S-LENGTH PIC S9(4) COMP. 49 S-VALUE PIC X(n).	
VARGRAPH and LONG VARGRAPHIC	01 G-VAR. 49 G-LENGTH PIC S9(4) COMP. 49 G-VALUE PIC G(n) DISPLAY-1.	

In COBOL, VARCHAR and LONG VARCHAR have the same format. VARCHAR and LONG VARCHAR are used for character data that varies in length. For VARCHAR, the maximum number of characters is 254. For LONG VARCHAR, the maximum number of characters is 32,767. You can define all data types at level 77 or level 01 except VARCHAR and VARGRAPHIC and their LONG variants. They must be at level 01 with sublevels 49, as shown above. GRAPHIC is used for Double Byte Character Set data.

SQL/DS supports the following data types in the FORTRAN format:

SQL Datatype	FORTRAN Equivalent
INTEGER	INTEGER INTEGER*4
SMALLINT	INTEGER*2
FLOAT	REAL REAL*4 REAL*8 REAL*16 DOUBLE PRECISION
CHAR(n)	CHARACTER CHARACTER*n

FORTRAN doesn't support the DECIMAL or GRAPHIC data types. However, it does support INTEGER, REAL, and DOUBLE PRECISION main variables for conversion to and from DECIMAL columns. FORTRAN doesn't support VARCHAR or LONG VARCHAR. However, it supports fixed-length CHARACTER main variables for conversion to and from VARCHAR and LONG VARCHAR columns. FORTRAN doesn't support VARGRAPHIC or LONG VARGRAPHIC at all.

Coding SQL Commands

At the start of every SQL program, you must place SQL statements that:

- Declare an SQL Communication Area (SQLCA).
- Declare special variables that SQL uses to interact with the host program.

The generalized COBOL structures you need to imbed SQL commands in your program take this form:

```
WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        .
        .
        .
       (Variable definitions used by SQL go here.)
        .
        .
        .
     EXEC SQL END DECLARE SECTION END-EXEC.
     EXEC SQL INCLUDE SQLCA END-EXEC.
        .
        .
        .
PROCEDURE DIVISION.
        .
        .
        .
     EXEC SQL command-name...      ...END-EXEC.
```

The generalized FORTRAN structures you need to imbed SQL commands in your program take this form:

```
EXEC SQL BEGIN DECLARE SECTION
    .
    .
    .
  (Variable definitions used by SQL go here.)
    .
    .
    .
EXEC SQL END DECLARE SECTION
EXEC SQL INCLUDE SQLCA
    .
    .
    .
EXEC SQL command-name...
```

Place the data description entries for all variables referenced in SQL statements in the SQL declare section. You can use the variables appearing in these SQL declare sections in regular COBOL or FORTRAN statements as well as in SQL statements. When referencing variables in SQL statements, the variable name must be preceded by a colon (:). When the same variable is referenced in an ordinary COBOL or FORTRAN statement, omit the colon.

Variables used in SQL statements can't be any of the following:

- Vector or array declarations
- A constant defined by a PARAMETER statement
- Any declarations that use expression to define the length of the variables
- Character variables declared with an undefined length such as CHARACTER*(*).

Logical Units of Work and Error Handling

The term **logical unit of work** means a sequence of SQL commands that SQL/DS views as a unit of consistency and recovery. (These commands can be mixed with non-SQL statements.) This concept is useful because SQL/DS can ensure the integrity of the data base. It does this by making sure that either **all** or **none** of the updates in a logical unit of work are done. For example, for system errors SQL/DS automatically restores all changes made during a logical unit of work. You can achieve this explicitly by using the **ROLLBACK WORK** command.

A logical unit of work begins with any SQL command and ends with a **COMMIT WORK** or **ROLLBACK WORK** command. If a system failure occurs before the explicit end of a logical unit of work, SQL automatically does a rollback of all the work from the start of the logical unit up to the point of system failure.

You must tell SQL/DS what to do for SQL errors. First you declare SQLCA through the **INCLUDE SQLCA** command. Then you code the appropriate **WHENEVER** commands at critical points in your program. The scope of the **WHENEVER** command is determined by its position in the program rather than its execution sequence. This is illustrated in an example below.

The SQLCA has two especially important fields: the SQLCODE and SQLWARN.

SQLCODE contains a code that indicates the result of each SQL statement. The value in SQLCODE summarizes the execution of your SQL statements:

- When the value is zero, the command has executed successfully.
- When the value is negative, an error condition has occurred.
- When the value is positive, a normal condition (for example, End-Of-File), or a warning condition is indicated.

You can test SQLCODE with the **WHENEVER** statement, which also indicates the action to take. The syntax of this statement is:

```
WHENEVER SQLERROR action
```

Possible actions are **CONTINUE** or **GO TO label**. When SQLCODE is 100, it indicates a NOT FOUND condition, which you can test by:

```
WHENEVER NOT FOUND action
```

SQLWARNING indicates a warning condition. SQLWARNING occurs when SQLCODE is greater than 0 but not equal to 100, or the SQL warning indicator, SQLWARN0, contains the value *W*. The syntax is:

```
WHENEVER SQLWARNING action
```

The normal action specified for SQLWARNING is **CONTINUE** or **GO TO label**.

In this COBOL example, the WHENEVER command causes a branch to ERRCHK when an error condition occurs (SQLERROR becomes negative) throughout the program. At ERRCHK, the WHENEVER is reset to CONTINUE during execution of the ROLLBACK WORK to prevent a failure during ROLLBACK from causing a program loop. After the branch back to DISPMENU the WHENEVER branch to ERRCHK is in effect again.

```
PROCEDURE DIVISION.
    .
    .
    .
    EXEC SQL WHENEVER SQLERROR GO TO ERRCHK END-EXEC.
    .
    .
    .
DISPMENU.
    .
    .
    .
ERRCHK.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    EXEC SQL ROLLBACK WORK END-EXEC.
    GO TO DISPMENU.
```

In this FORTRAN example, the WHENEVER command causes a branch to statement 90 when an error condition occurs (SQLERROR becomes negative) throughout the program. At statement 90, the WHENEVER is reset to CONTINUE during execution of the ROLLBACK WORK to prevent a failure during ROLLBACK from causing a program loop. After the branch back to statement 10, the WHENEVER branch to 90 is in effect again.

```
      .
      .
      .
      EXEC SQL WHENEVER SQLERROR GO TO 90
      .
      .
      .
10    LASTRC = ISPLNK ('DISPLAY','MENUPAN ')
      .
      .
      .
90    CONTINUE
      EXEC SQL WHENEVER SQLERROR CONTINUE
      EXEC SQL ROLLBACK WORK
      GO TO 10
```

Creating SQL/DS Tables

The table creation authority is RESOURCE. If you aren't sure that you have RESOURCE authority, speak to your data base administrator. You can create tables without RESOURCE authority in a PRIVATE DBSPACE created for you by a data base administrator.

A **DBSPACE** is a portion of the data base that can contain one or more tables and any associated indexes. Each table stored in SQL/DS is placed in some particular DBSPACE chosen by the creator of the table. The data base administrator defines DBSPACEs when the data base is generated. Additional spaces can be added later via the ADD DBSPACE function. Each DBSPACE remains as an unnamed "available" DBSPACE until it is "acquired" by means of an ACQUIRE DBSPACE statement. The acquiring user gives a name to the DBSPACE and defines certain characteristics for it (or allows default characteristics).

With SQL/DS you define new objects in the data base without stopping the system or calling special utilities. Your application program can create tables for storing and manipulating temporary results. It can then drop the table when it's no longer needed. You can also create indexes and drop table indexes as well as synonyms for table names. By using SQL Data Definition statements, you can accomplish these functions.

The table-id part of the **CREATE TABLE** statement specifies the table name. As a default, your table name is prefixed with your userid. The specifications for the table are pairs of column-names and data types with or without the qualifier NOT NULL. This qualifier tells SQL/DS not to allow null values in a particular column.

Any statement that later tries to put a null value in that column is rejected with an error code. The optional DBSPACE parameter lets you choose a specific data base space in which to create the table.

For example, the following statement can help you create a table called NAMELIST:

```
CREATE TABLE NAMELIST
          (FRSTNAME  CHAR(16) NOT NULL,
           LASTNAME  CHAR(16) NOT NULL,
           SERIALNO  CHAR(6) NOT NULL,
           AREACODE  CHAR(3),
           ZIPCODE   CHAR(5),
           PHNUMBER  CHAR(7))
        IN TEST.DBSP
```

Once a table is created, you can't change the data types of its columns or drop a column from the table. However, you can add new rows to the table using the **INSERT** command. For example:

```
INSERT INTO NAMELIST
   VALUES ('LEE', 'GREEN', '123456')
```

The command adds Lee to the first column, Green to the second column and 123456 to the third column.

You can also add new columns to a table using the **ALTER TABLE** command, or drop or delete a table using the **DROP TABLE** command. You must be the creator of the table or have data base administrator authority to delete a table.

For more information on data base spaces and SQL/DS, see *SQL/DS Application Programming.*

Querying SQL/DS Tables

When accessing SQL/DS tables, use cursor management routines. In general terms, a cursor is a pointer to the data base. The SQL **DECLARE** statements define a cursor by associating a name you choose with a query. The query may cause many rows to be returned from the data base. These rows are called the **active set** of the cursor. For example:

```
DECLARE CRSR CURSOR FOR        <---cursor-clause
SELECT FRSTNAME, LASTNAME      <---SELECT-clause
FROM NAMELIST                  <---FROM-clause
WHERE SERIALNO = :EMPSER       <---WHERE-clause
ORDER BY ZIPCODE               <---ORDER-BY-clause
```

In order to retrieve SQL data, you declare (name) a cursor (CRSR in this example) and associate with it a SELECT statement that describes the conditions and tables required for the retrieval. The SELECT statement must include a SELECT-list that specifies the columns (FRSTNAME, LASTNAME) required and a FROM-list that specifies the table(s) (NAMELIST) that contains those columns.

Optionally, a WHERE-clause may be used to filter the results. If it is not provided, all rows will qualify for the retrieval. Refer to "Defining Search Conditions" below for more detail.

The optional ORDER-BY-clause permits ordering the results of the query. Without it, the ordering is unpredictable.

After you issue the **DECLARE CURSOR** statement, you must open the cursor with an **OPEN** statement. For example:

```
OPEN CRSR
```

using the same cursor-name you specified in the **DECLARE CURSOR** statement.

Next you issue the **FETCH** statement. This tells SQL/DS to advance the cursor to the next row of the active set (result) and deliver the data into the main variables you specify on the **FETCH** statement.

The syntax of the **FETCH** statement is simple:

```
FETCH CRSR INTO :FNAME
```

In the **FETCH** statement, you must follow certain punctuation rules. Separate the main variables from each other with commas and precede each one with a colon. For example:

```
FETCH CRSR INTO :FNAME, :LNAME
```

When you're finished with a cursor, issue the **CLOSE** statement. For example:

```
CLOSE CRSR
```

Defining Search Conditions

To find particular items of data in SQL data bases effectively, you need to define **search conditions** in the WHERE-clause. These let you control row selection. A search condition is a collection of **predicates**. Each one specifies a test that SQL/DS applies to the rows of the table.

For example:

```
   WHERE SERIALNO = :EMPSER
```

causes SQL/DS to test the values in the SERIALNO column of each row of the NAMELIST table. SQL returns rows to the active set only when the SERIALNO value equals the value in the main variable EMPSER.

The criterion:

```
  SERIALNO = :EMPSER
```

is a predicate.

Along with column names, you can also use constants, variables, and any combination of these connected by arithmetic operators.

There are four arithmetic operators:

+	Addition
-	Subtraction
*	Multiplication
/	Division

You can use parentheses in an expression if you want to establish precedence among the operators. The default order of precedence is: negation, multiplication, division, addition, subtraction.

In addition to the equal sign, you can use:

¬=	Not equal to.
>	Greater than.
>=	Greater than or equal to.
<	Less than.
<=	Less than or equal to.

For example:

```
   PARTNO > 105
```

If you use variables in an expression, you must precede the variable name with a colon. This distinguishes it from a column name. Thus, the predicate:

```
SERIALNO > :EMPSER
```

means the value in column SERIALNO is greater than the value in variable EMPSER.

Conversely the predicate:

```
:EMPSER  >  SERIALNO
```

means the value in variable EMPSER is greater than the value in column SERIALNO.

You can also use constants within expressions, using any data types the language supports, but with some exceptions. See *SQL/DS Application Programming*.

You can also use the logical operator NOT to negate a predicate. For example:

```
AREACODE = 213 AND NOT ZIPCODE = 90023
```

You can connect predicates with the logical operators AND and OR:

```
AREACODE = 213 AND ZIPCODE = 90021 OR ZIPCODE = 90022
```

You can use this precedence rule with these operators: Apply the NOT first, followed by AND, followed by OR. In this above example the statement is true when AREACODE = 213 and ZIPCODE = 90021 **or** when ZIPCODE = 90022 (regardless of the value of AREACODE).

By using parentheses, you can override this order. If you want to select data only when AREACODE equals 213 and ZIPCODE equals **either** 90021 **or** 90022, you can code:

```
AREACODE = 213 AND (ZIPCODE = 90021 OR ZIPCODE = 90022)
```

Since the AND is evaluated before the OR, this is equivalent to:

```
ZIPCODE = 90021 OR ZIPCODE = 90022 AND AREACODE = 213
```

Additional Predicates

SQL provides four additional types of predicates you can use in search conditions. You can use them in addition to the standard ones that compare two expressions. These predicates are:

`BETWEEN` Determines if the value of an expression lies between the values of two other expressions. For example:

```
ZIPCODE BETWEEN :LIM1 AND :LIM2
```

This is equivalent to:

```
:LIM1 <= ZIPCODE <= :LIM2
```

`IN`	Lets you compare the value of an expression with a list of items. The predicate is satisfied if the expression equals any item listed. For example: `ZIPCODE IN (90021, :P2, :P3, :P4)`
`IS NULL`	Lets you explicitly look for null values in tables (empty fields) or exclude null values from consideration. For example: `ZIPCODE IS NULL`
`LIKE`	Lets you search for character string data that partially match a given string. For example: `FRSTNAME LIKE "%ANNE%"` This example is met by VALUES SUCH AS "ROXANNE," "ANNETTE," and "JANNER" as well as by "ANNE." The percent sign (%) represents a wild-card character and means any string of zero or more characters.

You can prefix any of these predicates with the logical operator NOT.

Built-In SQL Functions

SQL has built-in functions that you can use in expressions. All four functions have this format. For example, you can get the average of QUANTITY with the expression:

```
DECLARE CRSR CURSOR FOR
SELECT AVG(AGE)
FROM NAMELIST
```

These are the functions:

AVG	Computes the average value of the items specified.
MAX	Computes the maximum value of the items specified.
MIN	Computes the minimum value of the items specified.
SUM	Computes the sum of the values specified.
COUNT	Returns a count of the items specified.

Excluding Duplicates

The keyword **ALL** causes every value that satisfies the expression to be selected. This is the default. The keyword **DISTINCT** limits the selection to a single match.

For example, to get a list of different surnames, you'd use an expression such as:

```
DECLARE CRSR CURSOR FOR
SELECT DISTINCT LASTNAME
FROM NAMELIST
```

Manipulating Data in SQL/DS

There are SQL data manipulation statements that let you insert new rows into tables. You can also delete or update existing rows. Here are the three data manipulation statements:

1. **INSERT** lets you insert one new row into a given table. Also, by using the **SELECT** clause, you can insert several new rows selected or computed from other tables. You can insert data into any table you create. You can also insert data into another user's table if the table creator or your data base administrator gives you the INSERT privilege.

2. **DELETE** lets you delete one or more rows from a given table. However, first you must specify a selection criterion (WHERE clause). Otherwise, the **DELETE** statement deletes all table rows and sets a warning indicator (SQLWARN4). You can test the value of SQLWARN4 and, in case of error, issue the ROLLBACK WORK command.

   You can also delete the row that the current cursor points to by specifying WHERE CURRENT OF cursor-name. You can delete rows from any table you create and in another user's table. All you need is the DELETE privilege on that table.

3. **UPDATE** lets you change the value of one or more fields in one or more rows of a table. You can also change the value of one or more fields in one or more rows of a table by specifying WHERE CURRENT OF cursor-name. You can also update rows of a table you created. You can update other user's tables if you have the UPDATE privilege on the columns.

For more information on the **INSERT**, **DELETE**, and **UPDATE** statements, see *SQL/DS Application Programming*.

Creating Views in SQL/DS

SQL/DS can create **views** of a table. This is one of its most useful facilities. Views let you and other users see different presentations of the same data.

For example, if your NAMELIST table contains employee salaries, you'll want to restrict access to that data. Other users may need to see salaries but not addresses, and so on. Each user can have a different view of the data in the NAMELIST table. Each view appears to be a table and has its own name.

Views are based on tables, but views are not stored as physical tables. As such, they have some restrictions that real tables do not have. For example, some types of views cannot be updated. Refer to "Modifying Tables Through a View" in *SQL/DS Application Programming.*

You can create views by using the **CREATE VIEW** statement. (You must have SELECT privilege for the underlying table.)

In the following example we're selecting names and phone numbers of employees living in area code 707 for the view from the NAMELIST table:

```
CREATE VIEW AREA213 (FNAME, LNAME, EMPSER, PHONE) AS
SELECT FRSTNAME, LASTNAME, SERIALNO, PHNUMBER
FROM NAMELIST
WHERE AREACODE = 213.
```

The resulting view (which looks exactly like a table), is called AREA213. Its four columns have names distinct from the corresponding names in the NAMELIST table. If these names are not specified, SQL/DS takes them from the original table.

When you finish with a view, you can drop it:

```
DROP VIEW AREA213
```

There are certain restrictions on views. See *SQL/DS Application Programming.*

If you're programming in FORTRAN, skip to "Using SQL in FORTRAN Programs" on page 186.

Using SQL in COBOL Programs

This section briefly describes some rules you must follow for embedding SQL statements within a COBOL program. There are additional rules. For more information, see *SQL/DS Application Programming.*

Placement and Continuation of SQL statements

Place all SQL statements in columns 12 to 72. Place all declarative statements (like variable definitions) in the Data Division, within the File Section, Linkage Section or Working-Storage Section. All other SQL statements go into the Procedure Division. Continuation rules are the same for SQL statements as for all other COBOL statements.

Delimiting SQL Statements

Use delimiters to help SQL/DS distinguish SQL statements from COBOL statements. You must precede each SQL statement with **EXEC SQL** and terminate each one with **END-EXEC**. To conform with COBOL rules, you can place certain punctuation (like a period) after END-EXEC.

For example:

```
IF    FNTYPE = 1 THEN
      EXEC SQL INSERT INTO NAMELIST
      (SERIALNO, FRSTNAME, LASTNAME)
      VALUES (:EMPSER, :FNAME, :LNAME) END-EXEC
ELSE
      EXEC SQL UPDATE NAMELIST
      SET FRSTNAME = :FNAME, LASTNAME = :LNAME
      WHERE SERIALNO = :EMPSER END-EXEC.
```

SQL WHENEVER and **DECLARE CURSOR** statements shouldn't be the only contents of COBOL IF or ELSE clauses.

Using the Quote Parameter

If you use the COBOL compiler QUOTE option, you must also use the QUOTE option of the SQL/DS Preprocessor. Use a single quote (') to delineate string constants used in embedded SQL statements, regardless of the COBOL compiler QUOTE option.

Using the INCLUDE Command

When you want to include external secondary input, specify an INCLUDE statement at the point in the source code where you want to include the secondary input. For example:

```
EXEC SQL INCLUDE SOURCE1 END-EXEC.
```

Here SOURCE1 is the filename of a CMS file that is to be copied into the source program. It must have a filetype of COBCOPY. For a further discussion of the rules for using SQL statements in COBOL programs, see *SQL/DS Application Programming.*

If you're programming in COBOL, skip to “Preparation and Preprocessing a Program With SQL/DS” on page 187.

Using SQL in FORTRAN Programs

This section shows some of the rules you must follow for embedding SQL statements within a FORTRAN program.

The FORTRAN SQL Preprocessor

The FORTRAN SQL Preprocessor supports programs written for the VS FORTRAN compiler with the LANGLVL (77) option specified. Only FIXED FORM source statements are supported.

For a summary of the SQL statements supported by the FORTRAN Preprocessor, see *SQL/DS Application Programming.*

Placement and Continuation of SQL Statements

All SQL statements must be placed in columns 7 to 72. Columns 1 to 5 can also contain statement numbers, and columns 73 to 80 can contain sequence numbers and information.

The rules for the continuation of SQL keywords from one line to the next are the same as the FORTRAN rules for the continuation of words and constants. However, a SQL statement can use up to 124 continuation lines (for a total of 125 lines).

Embedding SQL Statements

You must precede each SQL statement in your program with EXEC SQL. No delimiter should be used at the end of any statement.

FORTRAN source statements can't be contained on the same line or within the same continued statement, except when a SQL statement is used as the imperative statement of a logical IF. Also, only one SQL statement can be contained in a single line, or within the same continued statement.

Using the INCLUDE Command

To include the external secondary input, use the SQL INCLUDE command and specify the filename of the file to be included. It must have a filetype of FORTCOPY. The file replaces the INCLUDE command in the source program. For example:

```
EXEC SQL INCLUDE SOURCE1
```

For more information about using SQL statements in FORTRAN programs, see *SQL/DS Application Programming.*

Preparation and Preprocessing a Program With SQL/DS

When your program contains SQL statements, it must be preprocessed by a SQL/DS preprocessor before it can be compiled. The preprocess step utilizes one of the SQL/DS preprocessors (a separate one is provided for each language) to convert your SQL/DS statements into valid (COBOL or FORTRAN) programming language statements.

After preprocessing and compiling, you must provide for linking your program with some special SQL/DS provided programs that are needed at the time of execution. This is done when you LOAD your application program. These SQL/DS provided programs, along with the SQL/DS EXECs that are needed to identify the SQL/DS data base and start the SQL/DS preprocessors, are stored on the SQL/DS "production" minidisk. You must access this disk in order to use SQL/DS in these procedures.

The procedures are illustrated in Figure 15. Following is more specific information on the steps identified:

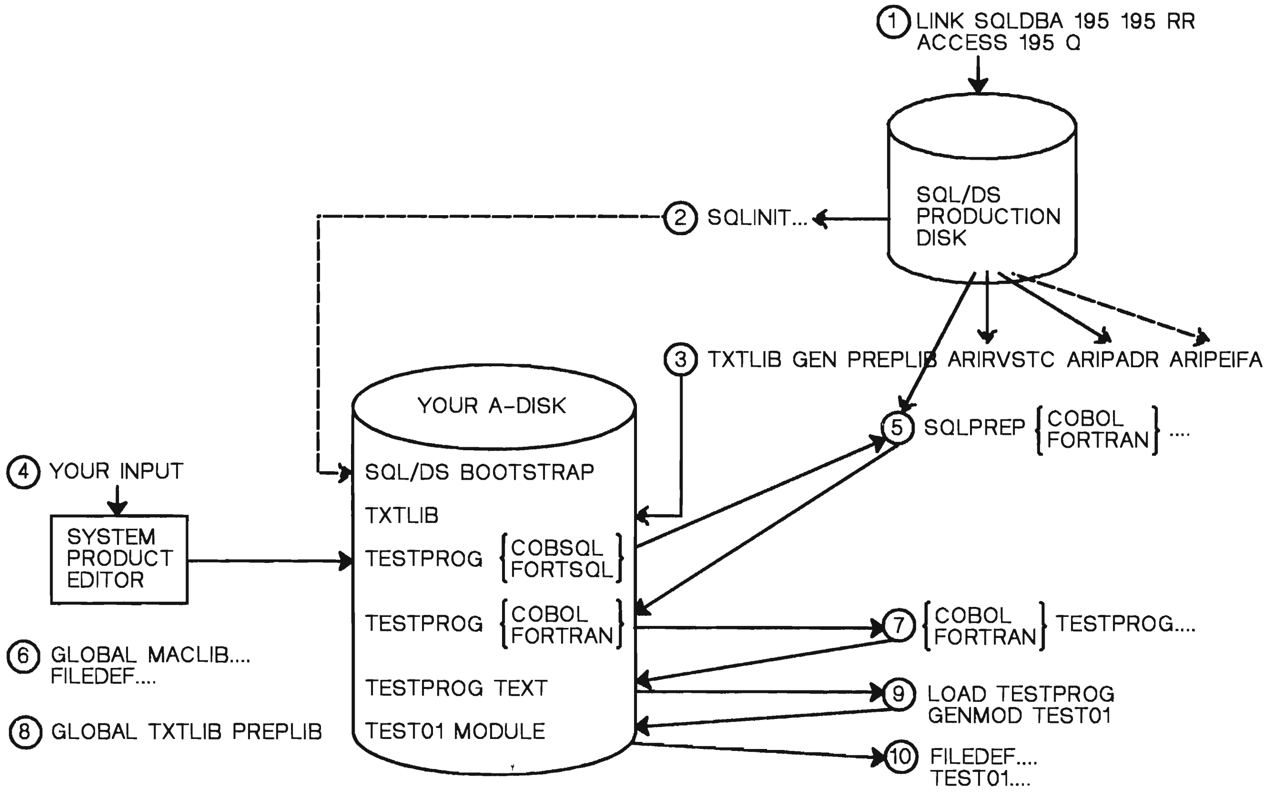


Figure 15. Creating an Executable SQL Program

1. Access the SQL/DS production minidisk:

```
LINK SQLDBA 195 195 RR
ACCESS 195 Q
```

The production mini-disk is established during the SQL/DS installation process. It contains the SQL/DS EXECs and programs required at execution time. These programs must be linked with your program in a later step.

2. Identify the SQL/DS data base. To do this, you will use the SQL/DS EXEC, SQLINIT. It names the data base and stores bootstrap information for that data base on your A-disk. Since this information is on your A-disk, you need only do this step once (even if you logoff), unless you subsequently need to change to another SQL/DS data base. An example of the use of this EXEC is:

   ```
   SQLINIT DBNAME(DBASE01)
   ```

   DBASE01, in this example, is the name of the SQL/DS data base selected.

3. Step 8 below requires that you have established a CMS TXTLIB that contains the execution time SQL/DS programs for linking with your program. This step builds that TXTLIB, containing the SQL/DS programs ARIRVSTC, ARIPADR, and ARIPEIFA. ARIPADR is required for COBOL, ARIPEIFA is required for FORTRAN, and ARIRVSTC is required for both languages. This step need only be done once (even if you logoff), since the TXTLIB is stored on your A-disk.

   ```
   TXTLIB GEN PREPLIB ARIRVSTC ARIPADR  (for COBOL)
                      or
   TXTLIB GEN PREPLIB ARIRVSTC ARIPEIFA (for FORTRAN)
   ```

4. You can use the System Product Editor to build your program. Give it a filetype of "COBSQL," if it is written in COBOL, or "FORTSQL," if it is written in FORTRAN. These are the filetypes that are required for the SQL/DS preprocessors in the next step.

5. The SQL/DS preprocessors are invoked through the SQL/DS EXEC, "SQLPREP." Following are examples of invoking the SQLPREP EXEC:

```
SQLPREP COBOL PREPPARM(PREPNAME=TESTPROG,QUOTE)
        SYSIN(TESTPROG) SYSPUNCH(TESTPROG) SYSPRINT(PRINTER)
(or)
SQLPREP FORTRAN PREPPARM(PREPNAME=TESTPROG)
        SYSIN(TESTPROG) SYSPUNCH(TESTPROG) SYSPRINT(PRINTER)
```

   - The language is specified as the first parameter. This selects the particular preprocessor and is followed by the parameters to that preprocessor.

   - PREPPARM has several subparameters. The main subparameter is PREPNAME. This is generally the same as the name that you have assigned to your program. For COBOL, you may want to use the keyword subparameter, QUOTE, to indicate that you are going to use the QUOTE option for the COBOL compiler. The QUOTE parameter (or APOST, the default) has no affect on the coding of SQL statements in the COBOL program, but informs the SQL preprocessor what to expect as delimiters for COBOL strings. These and other PREPPARM subparameters are explained more thoroughly in *SQL/DS Application Programming.*

- The SYSIN parameter specifies the filename of the input source program.

- The SYSPUNCH parameter specifies the filename of the output of the preprocess step, which is normally the same as specified for SYSIN. The default filetype assigned by SQL/DS is COBOL or FORTRAN, as required by the associated compiler.

- The SYSPRINT parameter specifies the filename (default filetype is LISTPREP) for receiving the printed output of the preprocessor. In this example, it is directed to the virtual printer, rather than a CMS file.

6. This step establishes the MACLIB and workfiles required by the particular compiler, for example:

```
       GLOBAL MACLIB COBOLVS CMSLIB
(or)   GLOBAL MACLIB FORTVS2 CMSLIB
       FILEDEF ...... (work files)
```

7. This step starts the appropriate compiler. Its input requires the filetype COBOL or FORTRAN and it produces a TEXT file.

```
COBOL TESTPROG
FORTVS2 TESTPROG
```

8. This step establishes PREPLIB as the GLOBAL TXTLIB for the LOAD step that follows. Step 3 created the PREPLIB that is used here.

```
GLOBAL TXTLIB PREPLIB
```

9. LOAD TESTPROG brings the new application text file into storage and links it with the required SQL/DS programs from PREPLIB. GENMOD TEST01 establishes a module on the A-disk for the application program and assigns to it the name "TEST01."

```
LOAD TESTPROG
GENMOD TEST01
```

10. After FILEDEFs, that may be required for the application program, the final step invokes the new program for execution.

```
TEST01
```

Summary

The Structured Query Language/Data System (SQL/DS) is a full scale data base management system integrated into the VM/SP environment. The Structured Query Language (SQL) handles SQL/DS data. SQL/DS also includes the ISQL facility that lets you enter SQL commands directly from your terminal. This lets you prototype applications that are to use SQL. ISQL also simplifies data handling by offering facilities for querying and manipulating data and writing reports. SQL/DS includes facilities for bulk-loading new data or data from existing systems into its relational data base.

Chapter 8: EXECs

An EXEC is a file of statements that are executed when you enter a single statement. You'll often need to perform a set sequence of VM commands, for example, when compiling and link editing a source program. You can group such sequences of commands in an EXEC file and control the execution of these statements by using additional EXEC statements.

In its simplest form, an EXEC file may contain only one record. In its most complex form it can contain thousands of records and resemble a complete program written in a high-level programming language.

There are three types of EXECs: CMS EXECs, EXEC 2 EXECs, and Restructured Extended Executor EXECs. All three types of EXECs are processed by the System Product Interpreter (or just "the interpreter"). We'll discuss each type of EXEC in order.

A Basic Exec

Here's an example of a simple EXEC procedure that you might use to relate file names in your application programs to their CMS file identifiers.

```
FILEDEF INPFL1 DISK INPUT DATA A
FILEDEF INPFL2 DISK MASTER DATA C
FILEDEF UPDT01 DISK TSTMSTR DATA A
FILEDEF CONSOL TERM
```

If you needed an EXEC like this, you'd create it using the editor. Let's say you stored it in a file with the file identifier DEFS EXEC A. Then, if you enter the command:

```
defs
```

each command line in the file is executed.

You must specify the filetype of all EXEC procedures as EXEC. CMS searches the disks currently accessed for a filetype of EXEC with a corresponding filename (DEFS in this case).

In an EXEC, CP commands are prefixed with **CP**, while the CMS commands are not prefixed at all. You can also call another EXEC from within an EXEC. In this case the prefix **EXEC** must be used. This is discussed in "Exec Arguments" on page 192. You can create EXEC files using CMS editors, by punching cards, or using CMS commands or programs.

The interpreter, which handles the execution of EXEC file contents, processes only the first 72 characters of each record in a fixed-length file and only the first 130 characters of each record in a variable-length file.

Profile Execs

The following EXEC links to a disk and defines its access order, sets up some characteristics for the terminal, and initializes some macro libraries.

```
CP LINK DEWEY 193 193 RR
ACC 193 B/A
CP SET EMSG ON
CP TERM HILIGHT ON
GLOBAL MACLIB OSMACRO CMSLIB
```

Such commands are typically issued at the start of every terminal session. If you give the EXEC a filename of PROFILE, it's automatically executed the first time you press the **ENTER** key after CMS is loaded.

Exec Arguments

An argument in an EXEC procedure is one of the special variable symbols &1 through &30 that are assigned values when an EXEC is invoked. For example, suppose the file COMPILE EXEC contains the following simple EXEC to compile a COBOL program:

```
CP LINK COBLIB 195 295 RR PASWD
ACC 295 E
COBOL &1
PRINT &1 LISTING
```

If you invoke the EXEC specifying the name of a COBOL source file:

```
compile testprog
```

the following procedure is executed:

```
LINK COBLIB 195 295 RR PASWD
ACC 295 E
COBOL TESTPROG
PRINT TESTPROG LISTING
```

The variable **&1**, which represents the first argument, is replaced by the token **TESTPROG** and is passed to the COBOL compiler. You can use up to 30 arguments by specifying &1 through &30.

The CMS EXEC File

You can create a special EXEC file called CMS EXEC by using the **LISTFILE** command with the EXEC option. Let's suppose you have a series of files on your disk with filenames beginning with the characters "PAY" and filetypes beginning with the character "D." If you enter:

```
listfile pay* d* a (exec
```

the usual **LISTFILE** display is placed in a file CMS EXEC. It has the format:

```
&1 &2 filename filetype filemode
```

Let's assume that after you issued the **LISTFILE** command shown, the CMS EXEC file contains:

```
&1 &2 PAYROLL   DATA       A
&1 &2 PAYDATE   DOCUMENT   A
&1 &2 PAYSLIP   DETAIL     A
&1 &2 PAY23UPD  D831102    A
```

If you now enter:

```
cms disk dump
```

the interpreter would execute the following commands:

```
DISK DUMP PAYROLL DATA A
DISK DUMP PAYDATE DOCUMENT A
DISK DUMP PAYSLIP DETAIL A
DISK DUMP PAY23UPD D831102 A
```

The arguments DISK and DUMP replace &1 and &2 when the file is executed. If only one argument is passed to an EXEC, the succeeding variables are set to nulls. For example, if you enter:

```
cms erase
```

these commands are executed:

```
ERASE PAYROLL DATA A
ERASE PAYDATE DOCUMENT A
ERASE PAYSLIP DETAIL A
ERASE PAY23UPD D831102 A
```

The CMS EXEC file is like any other CMS file. You can edit it, print it, sort it, and rename it. Each time you use **LISTFILE** with the EXEC option, a new CMS EXEC is created and an old one erased.

The EXEC 2 Processor

EXEC 2 programs and processing are broadly similar to CMS EXECs, with the following differences:

- There's no 8-byte restriction on token length. The words that comprise EXEC 2 statements can be up to 255 characters long.
- You can use EXEC 2 to issue commands to specified subcommand environments, such as the editor macro facility, as well as CMS and CP.
- EXEC 2 has extended string manipulation and arithmetic functions.
- You can define EXEC 2 subroutines and functions.
- EXEC 2 provides extensive debugging facilities.
- CMS user programs can manipulate EXEC 2 variables.

Although basic CMS EXECs can call EXEC 2 procedures and vice versa, the language statements can't be mixed within one EXEC. EXEC 2 coexists with the CMS EXEC processor program, and CMS examines the first statement of an EXEC file to determine which processor is required. If the first statement is **&TRACE**, the EXEC 2 processor is called to handle it.

With EXEC 2, you can assign variables, perform calculations, and control execution flow. Let's assume we have a set of files FILE1 TESTA, FILE2 TESTA, FILE3 TESTA, and so on. The following EXEC, called MULTIPRT, prints a specified range of these files a specified number of times. Printing starts and ends at specified files.

```
&TRACE
&ERROR &EXIT &RETCODE
* MULTIPRT EXPECTS THREE ARGUMENTS.
* IT DEFAULTS OTHERWISE.
&IF &N EQ 3 &GOTO -TSTARG
&TYPE INCORRECT NO OF ARGUMENTS SUPPLIED.
&TYPE DEFAULT VALUES HAVE BEEN ASSUMED
&ARGS 3 10 1
&GOTO -START
-TSTARG
&IF &2 ¬< &1 &GOTO -START
&PRINT INVALID ARGUMENT VALUES - PLEASE RESUPPLY
&READ ARGS
&GOTO -TSARG
-START &Y = &1
-DONEYET
&IF &Y > &2 &EXIT 0
&X = 1
&LOOP 2 UNTIL &X > &3
PRINT FILE&Y TEST A
&X = &X + 1
&Y = &Y + 1
&GOTO -DONEYET
&EXIT
```

If the user issues:

```
multiprt 11 14 3
```

the EXEC would execute three print commands for each of the following files:

```
FILE11 TEST A
FILE12 TEST A
FILE13 TEST A
FILE14 TEST A
```

This is the sequence of execution:

- The **&TRACE** control statement indicates to the system that this EXEC is written in EXEC 2.

- The **ERROR** control statement specifies that if any VM command results in a nonzero return code, the **&EXIT** statement is executed. In this example, the return code, indicated by the control word **&RETCODE**, is passed upon exit.

- An * indicates that a comment follows.

- **&N** is a special variable containing the number of arguments supplied. If the correct number is provided, then the EXEC procedure is routed to the label **-TSTARG**. Otherwise, **&TYPE** is issued. In EXEC 2, the restriction on word length is raised to 255. In CMS EXEC, you'd use the **&BEGTYPE** control word here.

- The **&ARGS** control statement is used to redefine any arguments that were entered. Thus, the default request is one copy of each of the files 3 through 10.

- At the label **-TSTARG**, the possibility of inconsistent arguments is tested using ¬ <, which means not less than. The **&PRINT** command means the same as the **&TYPE** command. The statement **&READ ARGS** issues a read to the terminal and assigns the tokens received to **&1, &2**, and so forth.

- The variable **&Y** is assigned the value of argument **&1**, in this case 11. You can include code statements (in this case **&Y** = **&1**) on the same line as labels.

- When **&Y** becomes greater than **&2** (14), the EXEC is ended with a return code of zero. This is the default and need not be specified.

- The variable **&X**, which counts the copies of each printed file, is set to **1**.

- The **&LOOP** control statement specifies that the following two commands are to be executed until **&X** is greater than the third argument (**3**).
- The VM **PRINT** command is issued. The file number **&Y** is joined with the word **FILE** to form the filename.
- When **&X** is incremented to 4, processing drops through to increment **&Y**. Also, the **&GOTO** control word jumps back to **-DONEYET** to check the value of **&Y**.

Note: If any of the files specified in the range to be printed do not exist, an error message will result.

Many EXEC 2 facilities are like those of a basic EXEC. Some control statements and special variables haven't been covered here. For full details on the EXEC 2 processor facilities, see *VM/SP EXEC 2 Reference.*

The Restructured Extended Executor Language

The third and most powerful language for use in EXECs is the Restructured Extended Executor language (REXX). It's a general-purpose, high-level language, not unlike PL/I, which is especially suited for prototyping and personal computing as well as handling EXEC command procedures.

REXX is a free-format language that can be coded to emphasize its structure, making it easier to read. There's no practical limit to the length of variable values, but variable names are limited to 250 characters. You can construct arrays using compound symbols such as:

```
NAME.X1.Y1
```

where **X1** and **Y1** can be the names of variables.

Although REXX is easy to use, its programs are executed using the interpreter; thus, it tends to use more computer time than an equivalent compiled language.

A Sample REXX Program

The following program illustrates some of the features of the REXX language:

```
/* The first line of the REXX EXEC
           must always be a comment              */
credits = 0
do until credits > 5
    a = random(1,9)
    b = random(1,9)
    say "What is "  a  "plus"  b  "?"
    pull answer  /* Place user's reply into answer */
    if answer = a + b
    then do
         credits = credits + 1; say "Correct."
         say "Your score is" credits
         end
    else
         say a "+" b "is" a+b
end
exit
```

This program repeatedly asks for the sum of two random numbers until it has accumulated six correct answers. Its processing is explained below:

- The comment delimiter **/*** on the first line indicates to CMS that this is a REXX program. This causes CMS to call the interpreter. The last comment line must end with ***/**.

- A value of **0** is assigned to the variable **credits**.

- The lines from **do until** to **end** are repeatedly executed as long as the value of **credits** doesn't exceed 5.

- Variables **a** and **b** are assigned random values in the range 1 to 9. The term **random(m,n)** is a built-in function of the interpreter; its arguments are the desired range in which the random number is to be generated. The interpreter has over fifty built-in functions, which are listed in *VM/SP System Product Interpreter Reference.*

- The instruction **say** writes the values of **a** and **b** to the console along with the literals "**What is**," "**plus**," and "**?**" in the order specified. The interpreter automatically inserts one space in the display between separate literals and/or variables. If more than one space is required, it must be incorporated into a literal (as, for example, in "**What is**").

- The instruction **pull** accepts the console reply into the variable **answer**. Comments in the executor language can be included on the same lines as program statements.

There are two forms of the **pull** instruction:

- the form **parse upper pull**, which is normally abbreviated to **pull**, translates everything read from the keyboard to upper case in the program.
- the form **parse pull** should be used if everything is required as is, without any translation.

- In the statement **if answer = a + b**, the item to the right of the = sign can be an expression, in this case **a + b**.
- When the test is true, more than one statement is to be executed. The **do...end** delimits these statements. If only one statement is to be executed, the delimiters are not required.
- So far there has been only one REXX language statement per line. A line-end is considered to be an implied delimiter. However, if more than one statement is to be placed on a line, the delimiter **;** can be used.
- If a correct answer requires no action, **if...then ; else...** would be incorrect. A semicolon doesn't cause a null instruction to be executed; the no-operation instruction **nop** would have to be used, as in **if...then nop else...**.
- When the test fails, the **else** portion is executed. You can include the value of expressions (for example, **a+b)** in the data to be displayed on the console.

For full details on interpreter instructions, see *VM/SP System Product Interpreter Reference.*

Issuing VM Commands

Although the program above contains only the REXX language statements, you can also use the language to control the execution of VM commands. The following program is an improved version of COMPILE EXEC, which was given in "Exec Arguments" on page 192. This EXEC accepts the file name as its argument. (The filetype is assumed to be COBOL.) Thus, if you had a COBOL program named TESTPROG on your A-disk to compile, you'd issue the statement:

```
compile testprog
```

For clarity, the VM commands are shown in upper case; the interpreter, however, doesn't differentiate between upper and lower case (except within strings, literals, etc.).

```
/* the executor language version of COMPILE EXEC */
SET CMSTYPE HT
arg a
Mainpart:
    signal on error
    COBOL a
    PRINT a LISTING
    SET CMSTYPE RT
    say a "Listing now printing."
    exit
    Error:
    rcsave = rc
    SET CMSTYPE RT
    say "Unexpected Return Code" rcsave "from command:"
    say "       " sourceline(sigl)
    say "at line number" sigl "."
    exit
```

Let's examine some of the features in this program:

- The **SET** command is used to halt typing (**HT**) of all details during the program's execution.
- The executor language command **arg** assigns the value of the argument supplied when the program is called to the program's variable a.
- The return code from commands is placed in the special REXX variable **rc**.
- In the executor language, a clause consisting of a single symbol followed by a colon is considered a label. The colon acts as an implicit terminator, so no semicolon is required, even when the label is followed on the same line by another statement. (In the example, the label **Mainpart:** is capitalized. It is not indented for clarity.)
- The instruction **signal on error** switches on a detector in the interpreter that tests the return code from every command. If a nonzero return code is encountered, the normal sequence of clauses is abandoned and execution transferred to a special label **Error:**. This detector can be switched off by issuing the instruction **signal off error**.
- The call to **COBOL** doesn't need to be prefixed by EXEC. This is true even when **COBOL** is an EXEC or EXEC 2 file, rather than another REXX EXEC. However, the interpreter won't implicitly let an executor language program EXEC call itself. If such a recursive call is necessary, the EXEC command can be invoked.

- In the **COBOL**, **PRINT** and **say** statements, the program variable **a** is evaluated to obtain the name of the COBOL program.

- When the **signal on error** detector encounters a nonzero return code, the interpreter assigns the line number of the failing command to the special program variable **sigl** and then transfers processing to the label **Error:**.

- The executor language function **sourceline (n)** returns the nth line in the source file. Here it's used with **sigl** to display the failing line of code. Its position in the display is indented by prefixing it with a literal of six spaces.

- The exit path from the program contains the CMS **SET RT** (resume typing command. This prevents the program from suppressing subsequent console displays.

Creating a System Product Editor Macro

You can write EXECs to be used with the System Product Editor that make creating or editing a file easier. These EXECs have a filetype of XEDIT rather than EXEC. Otherwise, they're like ordinary EXECs.

For example, this macro places continuation characters on specified lines in the correct column of COBOL or FORTRAN files. These EXECs are known as System Product Editor macros. You use them when you use the editor to create or edit a file. (For clarity, XEDIT commands are in upper case.)

```
/* Contchar macro */
Mainpart:
    SET MSGMODE OFF
    PRESERVE
    linlen = 1
    if arg() > 0 then linlen = arg(1)
    TRANSFER FTYPE
    pull filtype
    col = 72
    if filtype = 'COBOL'   then col = 7
    if filtype = 'FORTRAN' then col = 6
    SET TRUNC col
    SET ZONE col col
    C"/ /*/" linlen
    RESTORE
    SET MSGMODE ON
    exit
```

This is what the macro does:

- The message display option of the Editor is set off when you use the macro. Thus, its execution appears like a regular XEDIT subcommand.

- The **PRESERVE** command ensures that the settings of the various XEDIT variables, such as line length, are retained until the **RESTORE** command is executed.
- If an argument is supplied, the variable **linlen** is set to its value. Otherwise it remains equal to **1**.
- The XEDIT subcommand **TRANSFER** makes the filetype of the file being edited available on the console stack. Using the **pull** instruction, you can then assign the filetype to the variable **filtype**.
- The variable **col** is set variously to **6**,**7**, or **72**. This value is used to define the truncation column and then to set the zone.
- The **CHANGE** subcommand causes continuation characters to be included. The macro ends by resetting the environment to its original state.

Prototyping Interactive Applications

As mentioned in “Chapter 6: Using Dialog Managers” on page 141, you can conveniently use EXECs with ISPF applications.

EXEC 2 gives you the facility to call ISPF panel and variable services. Thus, you can write EXECs to invoke sequences of DISPLAY and SELECT services, and to handle related variables, before implementing the application in COBOL or FORTRAN. In EXEC 2, the format of a call to an ISPF service is as follows:

```
ISPEXEC service-name parameter1 parameter2 ...
```

EXEC 2 variables can be used anywhere in the statement as the service name or as a parameter. Each variable is replaced with its current value before execution of the **ISPEXEC** command. You can use parameter keywords wherever they apply. Otherwise, the parameters are positional. Here are some examples:

```
&SUBCOMMAND ISPEXEC DISPLAY PANEL( &PNAME)
&SUBCOMMAND ISPEXEC DISPLAY PANEL(MENUPAN)
```

In the first example, the EXEC 2 variable **&PNAME** is passed as a parameter. It's assumed to have a value **MENUPAN**. In the second example, this value is passed directly. You don't have to incorporate the keyword **PANEL**, since the parameter is in the first position. EXEC 2 requires that you precede ISPEXEC with a &SUBCOMMAND unless the statement:

```
&PRESUME &SUBCOMMAND ISPEXEC
```

is included in the procedure before executing the first ISPEXEC command. Some ISPF services allow dialog variables names to be passed as parameters. If you pass such names, don't precede them with an ampersand. For example:

```
ISPEXEC VGET XYZ
```

Here **XYZ** is the name of the dialog variable to be passed. The **VGET** service can also accept a list of variables passed as a single parameter. If you pass such a list, you must enclose it in parentheses. You must also separate the items with blanks or commas. For example:

```
ISPEXEC VGET (AAA,BBB,CCC)
ISPEXEC VGET (XXX YYY ZZZ)
```

ISPEXEC operations end with a return code in the same way as other routines do. Thus, you can use **&RETCODE** or **&RC** in EXEC 2 to test the success of the calls.

Here's an example of using ISPEXEC in an EXEC 2 procedure:

```
&TRACE OFF
&PRESUME &SUBCOMMAND ISPEXEC
CONTROL ERRORS RETURN
TBOPEN EMPLTBL
&IF &RC EQ 0 &GOTO -CONT1
TBCREATE EMPLTBL (EMPSER) (LNAME FNAME)
-CONT1
&F =
&EMPSER =
VPUT (F EMPSER)
DISPLAY PANEL (MENUPAN)
&IF &RC EQ 8 &GOTO -EXIT
&IF &F GT 4 &GOTO -EXIT
TBGET EMPLTBL
&IF &F EQ 1 &IF &RC NE 0 &GOTO -CONT2
&IF &F GT 1 &IF &RC EQ 0 &GOTO -CONT3
SETMSG MSG (MSG002 )
&GOTO -CONT1
-CONT2
&FNAME =
&LNAME =
-CONT3
SETMSG MSG (MSG001 )
&IF &F EQ 3 &GOTO -CONT4
DISPLAY PANEL (NAMEPAN)
&IF &F EQ 1 &GOTO -CONT5
&IF &F EQ 2 &GOTO -CONT6
&GOTO -CONT1
-CONT4
TBDELETE EMPLTBL
&GOTO -CONT1
-CONT5
TBADD EMPLTBL
&GOTO -CONT1
-CONT6
TBPUT EMPLTBL
&GOTO -CONT1
-EXIT
TBCLOSE EMPLTBL
&EXIT
```

This EXEC invokes a number of ISPF functions.

- **CONTROL ERRORS RETURN** tells ISPF to return to dialog processing when an error condition occurs (instead of terminating).
- **TBOPEN** tells ISPF to open the table EMPLTBL, if it exists. If the table doesn't exist, a non-zero return code is issued. The EXEC tests the return code, and if it is non-zero, the EMPLTBL table is created using the TBCREATE function.
- **VPUT** tells ISPF to initialize the panel variables specified (in this case, variables F and EMPSER). They are initialized to blanks prior to the panel display.
- **DISPLAY** tells ISPF to display the specified panel (MENUPAN or NAMEPAN) on the screen.
- **TBGET** tells ISPF to retrieve values from the EMPLTBL table.
- **SETMSG** tells ISPF to display the specified message on the next panel.
- **TBDELETE** tells ISPF to delete the current record (the one specified on the panel) from the EMPLTBL table.
- **TBADD** tells ISPF to add the current record to the EMPLTBL table.
- **TBPUT** tells ISPF to update the current record in the EMPLTBL table.
- **TBCLOSE** tells ISPF to close the EMPLTBL table.

If you want to run this EXEC, you will have to do the following:

1. Issue various FILEDEF commands to specify the ISPF libraries to be used. The necessary FILEDEFs to run the exec are listed in Appendix C.
2. Create a panel library called USERPAN MACLIB containing the MENUPAN COPY and NAMEPAN COPY files.
3. Create a message library, EXAMMSG MACLIB, containing definitions of messages MSG001 and MSG002. All panels and messages are described in Appendix C.
4. Invoke ISPF with the CMD parameter, instead of PGM when using an exec.

For more details on the use of EXECs with ISPF, see *ISPF Dialog Management Services – MVS, VM, and VSE.*

Using Execs with SQL/DS

You can easily use ISQL (Interactive SQL) when you access or update portions of tables on an ad hoc basis. When you do accesses or updates more routinely, you may want to group sets of ISQL commands into EXECs. An efficient way to execute these commands is to collect all the data you need, call ISQL, do the necessary commands, and re-exit to VM/SP.

VM/SP provides a stack for commands to be executed on a first-in first-out basis. You can place items onto the stack using the REXX command **QUEUE**.

Here's an example of an EXEC to change an entry in a phone list:

```
/* changnum exec  */
   say 'Supply last name'
   pull lnam
   say 'And now the initial'
   pull init
   say 'Enter new phone number'
   pull nnum
   queue COMMIT WORK
   queue "UPDATE PHONELIST -"
   queue "SET PHNUM = "nnum" -"
   queue "WHERE LNAME ='"lnam"' AND FINTL = '"init"'"
   queue COMMIT WORK
   queue EXIT
   exec  ISQL
```

In this example:

- When the necessary details are supplied to the EXEC, the interpreter puts the sequence of ISQL commands onto the command stack, using the **queue** command.
- The EXEC variables set to the supplied details are contained within the ISQL commands inside an inner set of quotes. (For clarity, the example shows them in lower case.)
- When the list is updated, the work is committed. Since we want to return to VM/SP after the operation, the ISQL command **EXIT** is stacked.
- Finally, the EXEC calls ISQL to start processing the commands queued on the stack.

Since the interpreter executes in the CMS environment, it isn't available while you're running ISQL. The stack gives you a way to transfer commands from one to the other. In this case you begin and end in CMS.

You can also build EXECs you expect to execute during ISQL sessions. Inside the EXEC you set a **RETURN** command as the first query in the stack. You no longer need the **QUEUE EXIT** and **EXEC ISQL** commands at the end of the EXEC. To start the EXEC during an ISQL session, you can enter “CMS” to get back into the CMS mode. You can then enter “CHANGNUM” (the name of the EXEC.)

When you supply the data and the EXEC ends, the **RETURN** command in the stack is executed. This takes you from CMS back into the ISQL environment. The rest of the stacked items (ISQL commands) are then processed. Since there's no **EXIT** at the end of the stack, you remain in the ISQL environment. Besides using this type of EXEC for less complex SQL table operations, you can use it for prototyping data base operations during design and development stages.

For more details on SQL, see “Chapter 7: Using SQL/DS” on page 167.

For more details on the use of EXECS with SQL, see *SQL/DS Planning and Administration — VM/SP*.

More Features of the Restructured Extended Executor Language

This section describes other REXX features that can help you to write more complex programs. We'll discuss substitution rules, compound symbols, and subroutines and functions.

Substitution Rules

When replacing the names of program variables with their assigned values, the interpreter doesn't check the substituted words to see if they're also variable names.

For example, the sequence:

```
food = meat
meat = steak
steak = sirloin
say "Buy" food
```

results in the display:

```
Buy MEAT
```

If the interpreter has not yet assigned a value to a program variable, it assigns the (capitalized) name of the program variable to be its value. Thus, in this example, the value of the variable **meat** is **MEAT**. The first instruction causes the variable **food** to have the same value. In the **say** instruction, **food** is evaluated and the result displayed.

If we rewrite the sequence using the REXX **value( )** function:

```
food = meat
meat = steak
steak = "sirloin"
say "Buy" value(food)||", that is," value(value(food))
```

the resulting display is:

```
Buy STEAK, that is, sirloin
```

Here the first call of the **value( )** function finds that **meat** has been assigned the value **STEAK**. In the second call, this value produces the literal **"sirloin"**. (The display would have capitalized **"sirloin"** if it were not assigned to the variable **steak** in the form of a literal.)

The || symbol causes concatenation without an intervening space; a comma follows the word **STEAK**. (You can insert a space before and after || without changing its effect.)

Compound Symbols

You can use compound symbols to build collections of variables, for example, to handle arrays of data.

Here is an example:

```
/* DAY exec */
day.1 = "Sunday"
day.2 = "Monday"
day.3 = "Tuesday"
day.4 = "Wednesday"
day.5 = "Thursday"
day.6 = "Friday"
day.7 = "Saturday"
do dayofmonth = 1 to 31
   dayofweek = (dayofmonth + 6)//7 + 1
   select
      when dayofmonth =  1 then th = "st"
      when dayofmonth =  2 then th = "nd"
      when dayofmonth =  3 then th = "rd"
      when dayofmonth = 21 then th = "st"
      when dayofmonth = 22 then th = "nd"
      when dayofmonth = 23 then th = "rd"
      when dayofmonth = 31 then th = "st"
      otherwise  th = "th"
  end
  say day.dayofweek dayofmonth||th "January 1984"
end
exit
```

In this example:

- Day-names are set up as an array called **day**. The **do** instruction repeatedly executes the entire program while incrementing the variable **dayofmonth** from **1 to 31**.

- The variable **dayofweek** (used to select the array elements) is cyclically assigned the values **1** through **7**. This calculation uses the operator //. (It means divide and return the remainder.) It has higher precedence than the symbol +, so the numerator is parenthesized to force it to be evaluated first. For a complete list of the Restructured Extended Executor language operators (comparative, arithmetic, etc.), see the

Syntax section of the *VM/SP System Product Interpreter Reference.* The Numerics and Arithmetic section contains full details of the arithmetic facilities of the interpreter.

- The **select** instruction evaluates each expression after the **when** keywords until it finds one to be true. It then executes the corresponding instruction. In the above example the value of the variable **th** is assigned appropriately. (If none of the **when** expressions is valid, the **otherwise** instruction is executed.)

- In the **say** instruction, the interpreter substitutes the value of **dayofweek** (for example, 1) into the compound symbol **day.dayofweek** to produce the derived name (day.1 for example). The result of the evaluation is the word **"Sunday"**.

The values of variables **dayofmonth** and **th** are concatenated (without an intervening blank) so that on the first pass the resultant display is:

```
Sunday 1st January 1984
```

You can use the compound symbol mechanism for arrays of more than one dimension by including extra periods. For example, the segments of a Rubik's cube (a three-dimensional array) might be called **cube.slice.row.column**, where **slice, row**, and **column** are variables having values 1 to 3.

Subroutines

The interpreter can call subroutines that are either in the same file as the main program or in a separate EXEC file. The program can pass up to ten arguments, and the subroutine can optionally pass back a result.

When the interpreter encounters the following instruction:

```
call mysubaa argument1 argument2
```

it searches the program for a label **mysubaa**, marking the start of the subroutine. If it doesn't find such a label, it continues searching for an external routine of the same name (since **mysubaa** isn't the name of a routine built in to the interpreter). To find more details on the interpreter's built-in and external functions, see *VM/SP System Product Interpreter Reference.*

Within the subroutine, you can retrieve the arguments by using the **arg** or **parse** instructions, which function in a parallel manner to the **pull** instruction described in "A Sample REXX Program" on page 197. Alternatively, you can use the built-in **arg()** function. For a full description of the **parse, arg**, and **pull** instructions, see the Parsing for PARSE, ARG, and PULL section in the *VM/SP System Product Interpreter Reference.*

The subroutine ends by executing either:

```
   return
```

or:

```
   return answervariable
```

The contents of the variable called **answervariable** are made available to the main program in a special variable called **result.**

Thus, the next instruction after a call might be:

```
   If result > 0 then ...
```

The variable **result** is one of the three special REXX variables that the interpreter can set; the other two are **rc** and **sigl**, which have been covered under "Issuing VM Commands" on page 198.

Functions

Functions are handled in a similar way to subroutines, with the following differences:

- They're executed as a function call instead of a **call** instruction.
- They must always return a result, even if only by issuing **return** " ".
- They don't use the special variable **result**.

Thus, the following instruction could execute the above subroutine as a function:

```
   if mysubaa(argument1,argument2) > 0 then ....
```

The following engineering program illustrates these items and some other points about subroutines and functions. The program is used to calculate the minimum amount of material needed to construct a closed box of fixed height and volume.

```
/* calculate minimum material required for box */
Mainpart:
    say "Enter required height"
    pull h
    if h <= 0 then
       do
       say "Invalid argument"
       exit
       end
    say "Enter required volume"
    pull volume
    if datatype(h,num) = 0 then
       do
       say "Invalid argument"
       exit
       end
    if datatype(volume,num) = 0 then
       do
       say "Invalid argument"
       exit
       end




    width = h
    oldm = boxmat(h,width,volume)
    width = width + 1
    call boxmat h,width,volume
    m = result
    if m > oldm then
       do until oldm <= m
          oldm = m
          oldlen = l
          oldwidth = width
          width = width - 1
          if width > 0 then
             m = boxmat(h,width,volume)
          end
    else
        do until oldm <= m
           oldm = m
           oldlen = l
           oldwidth = width
           width = width + 1
           m = boxmat(h,width,volume)
           end
    say "Required width =" oldwidth
    say "Required Length =" oldlen
    say "Minimum material =" oldm
    exit
Boxmat:
    procedure expose l
    arg h,w,v
    m = v/h
    l = m/w
    a = 2 * l * w,
      + 2 * l * h,
      + 2 * h * w
   return a
```

Here's an explanation of the above example:

- The label **Mainpart** is used to make the program more readable.
- The **height** and **volume** parameters are requested, and the values are accepted using the **pull** instruction.

  They're checked for validity by the built-in function **datatype.** The argument **num** indicates to the function that we're checking for numeric values other than zero. **Datatype** responds with **1** for numeric values. Otherwise **0** is returned. The result of the function is incorporated directly into the expression following the word **if**.
- As a starting point for the calculation, the variable **width** is estimated as equal to the height, **h**. The routine **boxmat**, which calculates the area of material required for a box of the given size, is now used to set the value of the variable **oldm**.
- The value of the variable **width** is increased by 1, and **boxmat** is executed again. In this case, **boxmat** is **call**ed as a subroutine. Its result is placed in the special variable, **result**. This contrasts with its previous use as a function when its result was directly incorporated into the expression.
- The variable **m** is now set equal to the value of **result**. Thus, we could just as easily have written:

  ```
  m = boxmat(h,width,volume)
  ```

  and not used the special variable **result** at all.
- Since we're calculating the *minimum* area of material required for a box, we must determine whether it's correct to increase the estimated width by 1. If the new area of material **m** is larger than the older value **oldm**, we're incorrect. We now repeatedly reduce the width estimate until a minimum value for the material is found.
- On the other hand, if we're correct in increasing the estimated width, the minimum area of material can be found by repeatedly calculating the area for increasing widths.

  In both cases, the **do until...end** instruction is used. In the case where the width decreases, a zero check for width is incorporated to prevent a zero value being passed to **boxmat**.
- When the new value of the area of material **m** starts increasing above **oldm**:
  - The minimum value has been found.
  - The **do until** loop terminates.
  - The required data is displayed.

- In **Mainpart**, **boxmat** is called, either as a function or as a subroutine, using the arguments **h**, **width**, and **volume**. By using an **arg** instruction, these values are assigned to the local variables **h**, **w**, and **v** within the routine.

  A **procedure** instruction is used to mask the variable names in the main program from those in the routine. This is necessary, as the variable **m** in the routine is not the same as **m** in the main program. However, the variable **l** (length) is required. The **procedure expose l** is used to let the main program "see" **l**, and only **l**.

- In calculating the area of material required, the expression for the value of **a** spills over onto more than one line. The comma is used to signify this to the interpreter.

- The **return** command indicates to the interpreter that **a** is to be assigned to the special variable **result** (if called as a subroutine), or inserted into the calling expression (if a function call was used).

Functions and subroutines that are in separate EXEC files don't have access to any of the main program's variables; thus, you don't need to use the **procedure** instruction above. However,you must pass all the required values as arguments.

You don't have to write subroutines in REXX. You can use EXEC, EXEC 2, or modules of code written in other languages, but they must all support the system interfaces that the executor language uses. For full details of these data, see *VM/SP System Product Interpreter Reference.*

Since the **procedure** instruction masks previous usage of variable names in the calling routine, you must use it when you make a recursive routine call.

For example, the following EXEC calculates the factorial of a number. You invoke the EXEC with a single argument, which is the number whose factorial is to be calculated.

```
/* calculate factorial */
START:
   arg x
   say 'x! =' factorial(x)
   exit
FACTORIAL:
   procedure
   arg n
   if n = 0
      then return 1
   return factorial(n-1) * n
```

The main routine calls the function **FACTORIAL**, which in turn calls itself, each time reducing **n** until it becomes zero. With each recursive call, the **procedure** instruction ensures that a new variable **n** is created.

Basic EXEC Language Facilities

Basic EXECs, can contain not only VM commands and calls to other EXECs, they can also contain keywords that begin with the character **&**. Many VM/SP systems have routines written in basic EXECs. It may therefore help you to understand some elements of basic EXECs. These can indicate control statements, built-in functions, and special variables, as well as the arguments covered in "Issuing VM Commands" on page 198.

Consider the following version of COMPILE EXEC, which was covered in "Exec Arguments" on page 192.

```
&CONTROL OFF NOMSG
&IF &INDEX LT 1 &GOTO -ERR1
GLOBAL MACLIB COBOLVS CMSLIB OSMACRO OSMACRO1
GLOBAL TXTLIB COBOLVS COBLIBVS CMSLIB
&IF &RETCODE NE 0 &EXIT &RETCODE
COBOL &1
&RC = &RETCODE
PRINT &1 LISTING
&TYPE &1 LISTING NOW PRINTING
&TYPE COMPILE EXEC COMPLETE
&EXIT &RC
-ERR1
&TYPE PROGRAM NAME NOT GIVEN
&EXIT
```

This CMS EXEC works as follows:

- The control statement **&CONTROL** sets the type of execution information displayed at the console. No execution messages or return codes are to be displayed here.
- The control statement **&INDEX** is a special variable that contains the number of arguments entered by the caller. (The control statement **&N** has the same function.) If no arguments are supplied, execution is to go to the label **-ERR1**.
- If the **GLOBAL** command fails, it results in a nonzero return code. The special variable **&RETCODE** contains the return code from the most recently executed CMS command. If it's nonzero, the control statement **&EXIT** is executed. This causes an immediate exit from the EXEC.
- When **COBOL** has been executed and the **PRINT** command issued, the **COMPLETE** and **PRINTING** messages are displayed on the console. The **&TYPE** statement isn't affected by the earlier **&CONTROL**, which only suppresses the display of the command lines being processed. The first **&TYPE** message contains the source program name in the form of the variable **&1**. When the second line has been typed, **&EXIT** is encountered and the EXEC terminates.
- If too few arguments are supplied, execution is routed to the label **-ERR1**, where the warning message is typed. No **&EXIT** is required here, since processing ends at the end of the EXEC.

In this section, some of the basic uses and facilities of EXECs have been explained. See *VM/SP CMS User's Guide* for tables of control statements and special variables and for more details on the CMS EXEC Processor and suggestions on writing EXECs. For the complete format and usage rules of each EXEC statement or variable, see *VM/SP CMS Command Reference.*

Using FILEDEF in EXECs

You can use the **FILEDEF** command to identify to VM the input and/or output files of an OS program. **FILEDEF** can be used in EXECs just like other VM commands, and can eliminate multiple lines of typing before a program is executed.

The following example demonstrates this by using the **FILEDEF** command inside a loop. This is possible because the ddnames and filetypes each contain a unique number as the last character.

```
/* set up payroll files */
Mainpart:
     say "Payroll Files - Weekly or Monthly (W/M)?"
     pull runtype
     say "How many Overtime files?"
     pull otime
     signal on error
     if runtype = 'W' then
         do
         FILEDEF INFILA C1 DSN STAFF.WEEKLY.PAYFILEA
         FILEDEF INFILB C1 DSN STAFF.WEEKLY.PAYFILEB
         end
     else if runtype = 'M' then
         do
         FILEDEF INFILA C1 DSN STAFF.MONTHLY.PAYFILEA
         FILEDEF INFILB C1 DSN STAFF.MONTHLY.PAYFILEB
         end
     else do
     say "Invalid reply - must be W or M - please restart"
     exit
     end
     do while otime > 0
         FILEDEF OTFIL||otime DISK OVERTIME DATA||otime B4
         otime = otime - 1
         end
     FILEDEF MASINP DISK STAFF MASTER1 B4
     FILEDEF MASOUT DISK STAFF MASTER2 A4
     FILEDEF CONSOL TERM
     exit
Error:
     say "Failure to execute FILEDEF command at"
     say "line number" sigl "Return code" rc
```

This EXEC first requests the type of **PAYFILE** and then the number of overtime files to be processed. Depending on the type, either weekly or monthly data sets are identified on the OS disk (in this example, the C disk).

The EXEC then uses **FILEDEF** to relate the internal ddnames of the form **OTFILn** (where n is a number) to the overtime files. These are in OS simulated data set format (on the B disk) and have CMS identifiers **OVERTIME DATAn B4** (where n is the same number used in the ddname).

The next three **FILEDEF** commands identify the master input and output files with ddnames MASINP and MASOUT respectively, and the VM terminal with the ddname **CONSOL**.

Use of MACLIBS and TXTLIBS in EXECs

CMS **MACLIBs** contain macro definitions and/or copy files. When you compile a source program with macro or copy definitions, you must be sure to identify the library containing the code before you invoke the compiler. Otherwise, the library isn't searched. The **GLOBAL** command identifies the libraries to be accessed and the order in which the compiler makes the search.

Here's an example of part of an EXEC.

```
/* compile a cobol prog */
Mainpart:
   signal on error
         .
         .
         .
   arg progname privlib
         .
         .
         .
   FILEDEF ......
         .
         .
         .
   GLOBAL MACLIB privlib OSMACRO OSMACRO1 TSOMAC
         .
         .
         .
   COBOL progname COBOL
         .
         .
         .
   say 'Any extra TXTLIBs required?'
   pull textlibr
         .
         .
         .
   if arg () = 0 then
       GLOBAL TXTLIB STDTXLIB
     else
       GLOBAL TXTLIB textlibr STDTXLIB
         .
         .
         .
   exit
ERROR:
         .
         .
         .
```

- The EXEC processes the compilation, link editing, and execution of a COBOL program. First you must supply two parameters: the program name (**progname**) and the programmer's private MACLIB (**privlib**). The **GLOBAL** command parameters are ordered so that the compiler searches the private library before the standard OS and TSO libraries.

- The **signal on error** command ensures that a nonzero return code from the call to **COBOL** causes execution to be routed to the label **ERROR:**. Otherwise, you must request any additional **TXTLIBs**.

- The **GLOBAL** command handles text libraries in a similar fashion to macro libraries. If you specify a private TXTLIB, it's incorporated in the **GLOBAL** command with the installation's library **STDTXLIB**.

Prototyping with REXX

REXX makes it easy for you to prototype algorithms before they're included in a larger compiled program. This procedure leads to faster program development, since design bugs are more quickly trapped without the need for multiple compilations. As mentioned in "The Restructured Extended Executor Language" on page 196, although the interpreter isn't executed as efficiently as compiled code, it takes less time to develop a program. Therefore, sizable savings result.

Here is an example:

```
/* Square Root Exec */
arg val
tol = 0.0001
old = 0
new = 1
count = 0
do while abs(old - new) > tol
       old = new
       work1 = old ** 2 + val
       work2 = 2 * old
       new = work1/work2
       say new
       count = count + 1
       end
say 'RESULT =' new 'CYCLES =' count
exit
```

This routine tests an algorithm for calculating square roots. Before incorporating it into a final compiled program (for example, in COBOL or FORTRAN), you can conveniently test its accuracy using a REXX procedure.

The procedure accepts the value whose root is required as an argument. As the main loop is executed, the current approximation to the root is displayed. At the end, the result is displayed, together with the number of cycles required to calculate it.

In prototyping, be careful when calling functions and subroutines. The mechanism the interpreter uses to pass arguments and results may not correspond to the mechanism of the compiled language that will eventually be used.

Summary

You often need to perform a set sequence of VM commands. You can group such sequences of commands in an EXEC file and control the execution of these statements by using additional EXEC statements. In its simplest form, an EXEC file may contain only one record. In its most complex form it can contain thousands of records and resemble a complete program written in a high-level programming language.

EXECs

There are three types of EXECs: CMS EXECs, EXEC 2 EXECs, and System Product Interpreter EXECs.

For full details on the EXEC 2 processor facilities, see *VM/SP EXEC 2 Reference*.

Chapter 9: Passing Commands and Data

This chapter describes:

- What a stack is and how it's used.
- What program linkages and return codes are provided by CMS.
- What parameter lists you use to issue CMS commands.
- What CMS macro structure is available to you.

Stacks

The CMS **program stack** is used to pass data between commands and programs.

CMS has storage area for terminal entries that exists for the life of the CMS session and is external to any programs or EXECs. This is called the console stack. The data elements within it include entries made from the terminal when the system isn't in VM read or CP read. These elements comprise the terminal input buffer. Other data elements within it are placed there deliberately by programs or EXECs. These are indistinguishable from terminal entries except by their placement; they reside in buffers within the program stack. Figure 16 on page 220 shows the elements of a console stack.

Terminal entries are always read on a FIFO (first in/first out) basis. If you make two separate entries, the first is processed first.

Program stack entries are under the control of the program or the EXEC. They may be queued FIFO or LIFO (last in/first out). If a program stacks two elements LIFO, the second one stacked is presented the next time that a read command is issued.

The buffers within the program stack are also in control of the program or EXEC. They can be used locally by one program, or globally to pass entries to another program.

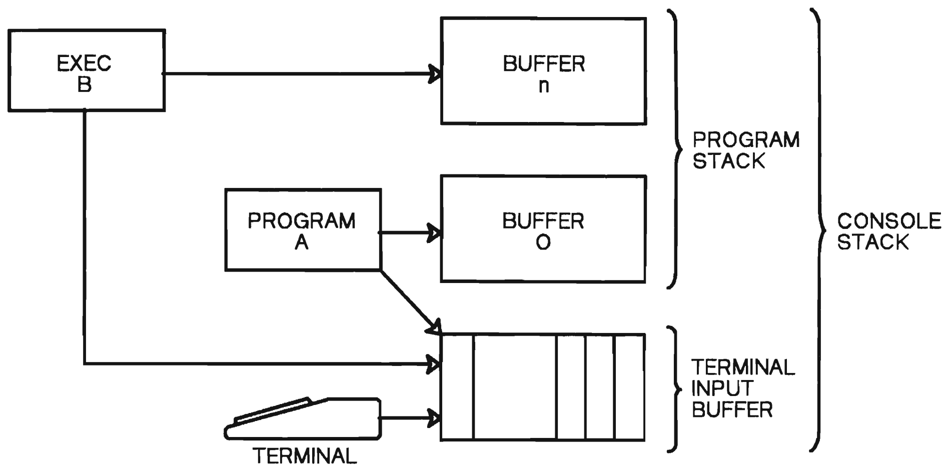


Figure 16. Elements of a Console Stack

Using a Program Stack Globally

Let's change the program you wrote so that it will automatically print out the time of day when it greets you by name. You can do this within the program, or you can do it externally.

You can use an EXEC (see "Chapter 8: EXECs" on page 191) to do this with no modifications to the program itself. You can also streamline the function by passing the parameters from the EXEC to the program using the program stack. If you do this, you don't have to respond to program prompts. When the program issues the read commands, it receives the stacked lines.

This EXEC performs such a function.

Use the editor to create this file on your disk as GREET EXEC.

```
&CONTROL OFF NOMSG
&STACK &1
&STACK &2
CP QUERY TIME
EXEC RUN TESTPROG
&EXIT
```

Then enter:

```
greet lee green
```

The system responds with the display:

```
TIME IS 11:45:38
      .
      .
      .
WELCOME TO CMS, LEE GREEN
```

This is the time of day message from CMS, followed by the familiar program output. The queueing function of the program stack passes data from the EXEC to your program.

Note: Sometimes you'll get the time message after the welcome message.

To see how the LIFO option works, modify the EXEC file so it looks like this:

```
&CONTROL OFF NOMSG
&STACK &1
&STACK LIFO &2
CP QUERY TIME
EXEC RUN TESTPROG
&EXIT
```

Make the same entry as before:

```
greet lee green
```

The system responds with:

```
TIME IS 11:49:57
      .
      .
      .
WELCOME TO CMS, GREEN LEE
```

The greeting now has the last name first.

Using a Local Stack

You use a local stack when your stacked messages apply only to your program. Assume you want to sort a file in nickname order before it's printed, but you don't want to change the original file. You can do this with the **SORT** command. This command is different from the editor **SORT** command you learned in "Manipulating Data" on page 60. The **SORT** command creates a new, and temporary, file. It also prompts for sort fields. You stack the answer in advance, therefore you don't have to respond to the prompt. However, the **SORT** command can fail before it issues the read for this information, so you'll clear out the buffer before exiting.

Note: This EXEC expects to find a file called DATA FILE A.

This is the code:

```
/* Sort and print */
MAKEBUF
QUEUE "1 8"
SORT DATA FILE A WORKDATA TEMP A
if rc ¬=0 then do
      say "unexpected return code",
      rc "from sort command"
      end
else PRINT WORKDATA TEMP A
DROPBUF
exit
```

This program sorts the file DATA FILE A to a workfile, and prints it. If a problem occurs in the **SORT** command, it deletes its own stacked message. It doesn't affect any following stack activity.

Figure 17 on page 223 illustrates how local buffers affect the system when you do the local stack example. The following takes place:

1. Your program gets control.

2. You stack the message for the CMS sort.

3. You issue the **SORT** command, transferring control.

4. Sort reads from the console stack, and performs the sort.

5. Your program performs the final steps, then exits.

6. CMS regains control.

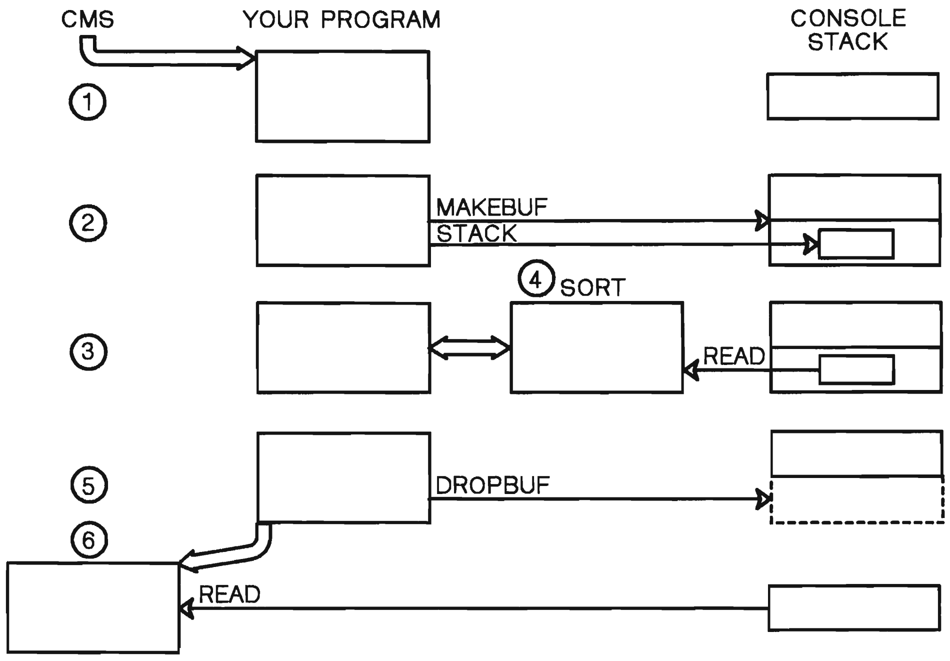


Figure 17. Example of Local Stack Usage

Manipulating the Program Stack

Some commands that put data into a program stack are:

- For REXX:

  QUEUE The data is queued FIFO.

  PUSH The data is pushed LIFO.

- For EXEC or EXEC2:

  &STACK FIFO The data is queued FIFO.

  &STACK LIFO The data is pushed LIFO.

Some commands that read data from a program stack are:

- For Restructured Extended Executor language:

  PULL The next item is read.

- For EXEC or EXEC2:

  &READ The next item is read.

The order in which stacked items are retrieved is determined at the time they're placed in the stack.

Using Program Stacks

The best way to make use of the program stack is to maintain control of it with the **MAKEBUF** and **DROPBUF** commands. The **MAKEBUF** command creates a new buffer within the program stack. The buffer number for the new buffer is returned in register 15 by CMS.

Note: If an &ERROR statement is in effect in an EXEC that invokes this command, the return code causes it to execute. Therefore, it's important to ensure that no &ERROR statement is in effect at the time.

After the **MAKEBUF** command is issued, you can determine how many entries are already on the program stack by issuing the **QUEUED()** function. This instruction returns the number of entries in the stack. (A similar function is available with the **SENTRIES** command.) Here again, the value is returned in register 15. Be careful not to create an invalid execution of the **SIGNAL ON ERROR** statement. The result of the **QUEUED** or **SENTRIES** command can be used by the program as a processing cutoff to avoid using any stack elements from another program or the terminal input buffer.

Place entries in the stack with the **QUEUE, PUSH**, or **STACK** commands. Retrieve them with the **PULL** or **PARSE PULL** commands, or any CMS commands that access the stack elements.

Program Linkages and Return Codes

If you write an assembler language subroutine, you need to know about the CMS linkage conventions, and the CMS parameter communication architecture. The linkages provided are the base address of your program, the return address, and the return code. The passed parameters can be in two formats:

1. **Tokenized Parameters** These are left justified, blank-padded, and truncated to eight bytes each. Each of the strings you enter, including the name of the module, is considered a parameter. A parenthesis is also considered a parameter. Blanks and parentheses are the delimiters. Tokenized parameters are always passed to the program, regardless of the method by which it's called.

2. **Extended Parameter Lists** These supply the data exactly as entered. Extended parameters are only passed to the program if it's called from the terminal or by an EXEC2, or Restructured Extended Executor Language EXEC file. You can use this form if more than one argument string is to be passed to the EXEC, or the EXEC is being called as a function.

Certain registers are reserved for this specific usage in CMS. These are:

Register 0	is a pointer to data describing the extended parameter list (**Plist**).
Register 1	is a pointer to the tokenized parameter list.
Register 13	is a pointer to the register save area.
Register 14	is a return address to be branched to at exit.
Register 15	is a return code given to you, or set by you before return.

Linkage Registers

Your program gets control with its base and return addresses set up in registers. These are the most important of the linkage registers. You don't need to set up your own base register; it's provided. You must return to the return address or the CMS session will end.

The base address is passed to your program in each of two registers: register 12 and register 15. CMS calls destroy the contents of register 15. To maintain addressability, use register 12 as a base. CMS routines restore register 12 before returning control to you. Save register 14 on entry into your program, and restore it on exit. CMS routines use register 14 but don't restore it.

This is a program that conforms to these linkage conventions. It's provided here as a sample of the minimum structure that a program should have:

```
PROGRAM CSECT
    USING PROGRAM,R12    ESTABLISH ADDRESSABILITY
    ST    R14,SAVRET     SAVE RETURN ADDRESS IN R14
      .
      .
      .
    L     R14,SAVRET     LOAD RETURN ADDRESS
    BR    R14       RETURN
SAVRET   DS    F
```

Return Codes

Your program sends status information to the caller using a return code. This return code is in register 15. Other programs and EXECs inspect this register when control returns to them, and take appropriate action. If a terminal input calls you, the return code is displayed by CMS. Be sure to conform to the return code conventions. On normal exit, set register 15 to zero. On an error exit, set register 15 to some non-zero value.

Various return codes have taken on special meanings through usage. (See *VM/SP CMS Command Reference* for a further discussion of these codes.) If you plan to test for one of these errors, it's better to conform to the established code. This makes it easier for others to understand your program's ending status.

Let's assume that your program has a special error exit that it uses if a certain file isn't found. The standard error return code for this condition is 28. Modify the program to look like this:

```
PROGRAM  CSECT
     USING PROGRAM,R12    ESTABLISH ADDRESSABILITY
     ST    R14,SAVRET     SAVE RETURN ADDRESS IN R14
       .
       .
       .
EXITOK    DS    0H
     L     R14,SAVRET     LOAD RETURN ADDRESS
     LA    R15,0         SET NON-ERROR RETURN CODE IN R15
     BR    R14           RETURN
     SPACE 1
EXITNG    DS    0H
     L     R14,SAVRET     LOAD RETURN ADDRESS
     LA    R15,28        SET NON-ERROR RETURN CODE IN R15
     BR    R14           RETURN
SAVRET    DS    F
```

If this program had executed without errors and returned to CMS through the label EXITOK, the CMS ready message would have read:

```
Ready;
```

If it hadn't found the file and exited through EXITNG, the ready message would read:

```
Ready(00028);
```

This is recognized as an unsuccessful program ending, caused by a file not found.

In "Chapter 3: Using the System Product Editor" on page 39, you invoked the editor to create your test program. The command you entered was:

```
xedit testfile fortran a (noprof
```

This entry is made from your terminal, so both the extended and tokenized forms of the parameter list were passed to the module called XEDIT. This is an example of what the data looked like to that module:

- Register 1 contained the address of CMNDLIST.
- CMNDLIST had been set up like this:

```
CMNDLIST DS   0D
     DC   CL8'XEDIT'
     DC   CL8'TESTFILE'
     DC   CL8'FORTRAN'
     DC   CL8'A'
     DC   CL8'('
     DC   CL8'NOPROF'
     DC   XL8'FF'
```

This is the tokenized parameter list:

- Register 0 pointed to EPLIST.
- EPLIST had been set up like this:

```
EPLIST    DC   A(CMDSTART)
     DC   A(ARGSTART)
     DC   A(ARGEND)
     DC   A(0)
```

The extended parameter list referenced by EPLIST is:

```
CMDSTART DC   C'xedit'
ARGSTART DC   C'testfile fortran a (noprof'
ARGEND      EQU  *
```

The module XEDIT used this data to initiate your session with the file and options you wanted. You can use the Plist in the same way, when you want your program to process a similar input list.

Using Parameter Lists to Issue CMS Commands

You can issue a CMS command from your program by using a parameter list (Plist). Let's assume that this program updates a CMS file. You want to print the file (in upper case), and you don't care about errors in the CMS routine. You can issue the CMS print command from your program.

To do this you would set up the parameter list, load its address into R1, and call CMS with an SVC 202 that is followed by an adcon '1'. The program looks like this:

```
PROGRAM  CSECT
     USING PROGRAM,R12     ESTABLISH ADDRESSABILITY
     ST    R14,SAVRET      SAVE RETURN ADDRESS IN R14
       .
       .
       .
EXITOK    DS    0H
     LA    R1,PRINTIT      LOAD PLIST ADDRESS
     SVC   202            CALL CMS
     DC    AL4(1)         IGNORE ERRORS
     L     R14,SAVRET      LOAD RETURN ADDRESS
     LA    R15,0          SET NON-ERROR RETURN CODE IN R15
     BR    R14            RETURN
     SPACE 1
EXITNG    DS    0H
     L     R14,SAVRET      LOAD RETURN ADDRESS
     LA    R15,28         SET NON-ERROR RETURN CODE IN R15
     BR    R14            RETURN
SAVRET    DS    F
PRINTIT  DS    0D         ALIGN PLIST TO DOUBLEWORD BOUNDARY
     DC    CL8'PRINT'      COMMAND
     DC    CL8'DATA'       FILENAME
     DC    CL8'FILE'       FILETYPE
     DC    CL8'*'          FILEMODE
     DC    CL8'('          OPTION DELIMITER
     DC    CL8'UPCASE'     UPPERCASE OPTION
     DC    8X'FF'          FENCE END OF PLIST
```

This program issues a CMS print command when the job is successfully completed.

Using CMS Macros

You can invoke CMS services from the assembler program using **Plists** or **macros**. Heavily formatted Plists can be cumbersome. To streamline such transactions, CMS provides a group of macros. These macros let you manipulate files, perform I/O functions, and communicate with VM in a simple way.

- The file control macros permit you to access CMS files easily. Most of the file definition information is provided by defaults or is retrieved from the file by CMS itself.
- The CMS file management function uses a **file system control block (FSCB)**. This block contains all the information necessary to CMS. It serves as a repository for record pointers and other like data. CMS can automatically create the FSCB from data you provide in the form of macro options. Or, you can create the FSCB and refer to it by label in your macros.
- VM provides two macros for creating an FSCB:

FSCB	creates a file system control block for a CMS disk file.
FSCBD	creates a DSECT for the file system control block.

- Two macros can be used to open and close CMS files. While it isn't always necessary to use these macros, it's good practice. Future changes in the way the program is invoked, or in its internal file manipulation, may make them necessary. To track down their omission at that time is time-consuming. These macros are:

  FSOPEN opens a CMS disk file for input or output.

  FSCLOSE closes a CMS disk file.

- Reading from and writing to a CMS file is done using the following macros. They can be coded to point to a FSCB, or to provide the file identifying information within themselves. They have standard defaults for the various options, which let you code them easily.

  FSREAD reads from a CMS file into the program I/O buffer.

  FSWRITE writes to a CMS file from the program I/O buffer.

- Three macros perform various functions using the FSCB and standard defaults:

  FSSTATE retrieves information about the status and format of a CMS file.

  FSERASE erases a CMS file.

  FSPOINT resets the write or read pointers for a file.

- The I/O control macros cover all the normally needed operations for unit record, terminal and tape devices. Three macros perform unit record device I/O:

  PRINTL writes a line to a virtual printer.

  RDCARD reads a card from a virtual reader.

  PUNCHC punches a card on the virtual punch.

- The macro for full-screen I/O:

  CONSOLE performs 3270 I/O operations, including building the CCW, issuing the DIAGNOSE code X'58' or SIO instruction, waiting for the I/O to complete and checking any error status from the device.

- Four macros read and write to terminals one line at a time:

  LINERD allows users to read from a specified virtual screen.

  LINEWRT allows users to write to a specified virtual screen.

RDTERM reads a line of output from a terminal.

WRTERM writes a line of output on a terminal.

- This macro lets you suspend program execution until the terminal activity has been completed.

  WAITT causes the program to wait until pending terminal I/O is finished.

- Four tape macros perform normal I/O operations and provide certain label processing:

  TAPECTL positions the tape (rewind, backspace, etc.).

  TAPESL processes HDR1 and EOF1 tape labels.

  RDTAPE reads a record from the tape drive.

  WRTAPE writes a record to the tape drive.

- LINEDIT is used to communicate CP commands to VM:

  LINEDIT compiles, formats, and displays a message on your terminal, or presents it to CP as a CP command.

- The following macros are used for interrupt trapping:

  HNDEXT traps external interrupts for internal program handling.

  HNDINT traps interrupts for a specified I/O device.

  HNDSVC traps interrupts for a specified supervisor call.

  WAITD causes the program to wait until the next interruption occurs on the specified device.

- This macro selects relocatable and nonrelocatable members of the called files at linkage exit time:

  COMPSWT directs the CMS LOAD command to load non-relocatable modules.

- The following macro provides register equates.

  REGEQU generates a standard equate list for general, floating point, and extended control registers.

See *CMS Command Reference* for further information about the use of macros and their formats.

Summary

In this chapter, we discussed what a **program stack** is and how it is used. We discussed **program linkages**, **return codes**, and how to use **parameter lists** to issue CMS commands. We also discussed using **CMS macros** to manipulate files, perform I/O functions, and to communicate with VM.

Chapter 10: Testing and Debugging Programs under VM/SP

This chapter describes testing and debugging facilities that VM provides. It covers the **TESTCOB** and **TESTFORT** commands, **VS FORTRAN Interactive Debug**, **VS FORTRAN Version 2 Interactive Debug**, the dialog and testing service of ISPF, and the use of SQL/DS for data base prototyping and testing. It also summarizes the run-time debugging facilities of VM, which provide access to general registers, main storage, and control words, as well as trace options and dump control.

The use of **TESTCOB** and **TESTFORT VS FORTRAN Interactive Debug**, and **VS FORTRAN Version 2 Interactive Debug**, under CMS is broadly similar to their use elsewhere (for example, TSO), although, there are some differences under CMS.

Certain low-level facilities of VM/SP require you to be familiar with the structure of the system. If you wish to use them, you should have *IBM System/370 Principles of Operation* available for use as a reference.

| *Note: If you are testing and debugging a program that issues vector*
| *instructions, refer to Chapter 11 for more information.*

Interactive Debug

The Interactive Debug products contain a command and a set of subcommands that aid you in diagnosing and solving problems in your programs.

The facility lets you:

- Stop and start the program as it runs.
- Examine and change values of variables.
- Trace program transfers.
- Track frequency of execution of statements.
- Locate errors and correct them.
- Test the code and improve its efficiency.

COBOL Interactive Debug

You can use COBOL Interactive Debug under CMS to debug any COBOL program compiled with the TEST option. Although with the **RUN** command, you can compile, load, and execute a program in one step, to use Interactive Debug you should use the **COBOL** command with the keyword TEST to compile your program. You can use other COBOL command parameters. However, some of them may have a slightly different effect. For details, see *IBM OS COBOL Interactive Debug Terminal User's Guide and Reference.*

After you've compiled the program, you have a stored, TEST-compiled TEXT file ready to be processed with Interactive Debug. Before proceeding, however, you must issue a **GLOBAL** command for the TXTLIB files with the COBOL library routines and any previously compiled routines called by your program.

When running under Interactive Debug, you must define all files by using the **FILEDEF** command. This includes SYSIN, SYSOUT, and SYSPUNCH, if needed, and any input or output files your program uses. The Interactive Debug also requires a **debug** file for its own use.

You can define this file by issuing a command of the form:

```
filedef d disk mycobfil sysut5
```

The use of the CMS commands **GLOBAL** and **FILEDEF** are described in "Chapter 4: More about Compiling and Running a Program" on page 105.

You can now issue the **TESTCOB** command. It's like **EDIT** in the way it lets you enter subcommands to manipulate data and monitor the execution of your program.

TESTCOB provides several subcommands that let you control the execution of your program:

GO starts the execution of the program. Until you actually issue the **GO** subcommand, TESTCOB waits.

If you want to return to TESTCOB (rather than continuing with the execution of the program), press ATTN twice. The system displays TESTCOB to indicate that you can enter subcommands.

AT specifies a line number where program execution is to stop. When you specify **AT**, you can specify variables whose values are to be displayed.

LIST specifies a variable whose value you want to be displayed.

END ends the execution of TESTCOB. (Pressing the ATTN key twice does the same thing.)

Example

Here's an example of the use of TESTCOB. Compile the following program **mycobfil** for testing using the TEST option.

Note: This is like the program we entered in "Creating a COBOL File" on page 19, but it contains an intentional mistake. The word AFRST in line 170 should be ALAST.

```
010    IDENTIFICATION DIVISION.
020    PROGRAM-ID. MYPROG.
030    ENVIRONMENT DIVISION.
040    DATA DIVISION.
050    WORKING-STORAGE SECTION.
060    77  FNAME     PIC X(22) VALUE "ENTER YOUR FIRST NAME.".
070    77  LNAME     PIC X(23) VALUE "AND NOW YOUR LAST NAME.".
080    01  ANSWR.
090        05    ANSLT         PIC X(16) VALUE "WELCOME TO CMS, ".
100        05    AFRST         PIC X(8)  VALUE SPACES.
110        05    FILLER        PIC X     VALUE SPACES.
120        05    ALAST         PIC X(8)  VALUE SPACES.
130    PROCEDURE DIVISION.
140        DISPLAY FNAME UPON CONSOLE.
150        ACCEPT  AFRST FROM CONSOLE.
160        DISPLAY LNAME UPON CONSOLE.
170        ACCEPT  AFRST FROM CONSOLE.
180        DISPLAY ANSWR UPON CONSOLE.
190        STOP RUN.
```

Now you issue this command:

```
testcob mycobfil (myprog d)
```

where:

MYCOBFIL is the name of the TEXT file resulting from the COBOL compilation.

MYPROG is the name of the program (as specified in the PROGRAM-ID statement).

D is the ddname of the debug file set up in the **FILEDEF** command.

After you enter this command, the system displays **TESTCOB**. This means that the system is ready to accept a command from you. Now you can, for example, say that you want to see the value of AFRST at line 190. Once you set this, you can start your program executing.

The subcommands go like this:

```
TESTCOB
at 160 (list afrst)
TESTCOB
go
```

The effect of this is to continue execution of the program to line 160. The system then displays a message in the following form:

```
AT MYPROG.160.1
 07A855 100       02  AFRST   A     FRED
TESTCOB
```

You can interpret the response as follows:

- TESTCOB halts the program at the first verb on source line number 160.
- The variable AFRST, defined on source line number 100, has a main storage location of 7A855 and a (normalized) level number of 02.
- AFRST is alphabetic (A) and currently has a value of "FRED."

You can request display of more than one variable and enter multiple subcommands. For example, if you enter:

```
at 190 (list afrst;list alast;go)
```

the display might look like this:

```
AT MYPROG.190.1
 07A855 100      02 AFRST   A    SMITH
 07A870 120      02 ALAST   A
PROGRAM UNDER TESTCOB ENDED NORMALLY
TESTCOB
```

In this example, the following occurs:

- The system displays the values of both variables. The value of ALAST remains unmodified (i.e., spaces).
- The program starts executing again and it displays the welcome and request messages again.

When you find that ALAST is still set to spaces, you might need to alter its value before proceeding with the test. You can use the **SET** subcommand to do this:

```
set alast = jones
```

You might, however, want to examine the error by looking at the source line that was supposed to change it. In TESTCOB mode, issuing the subcommand:

```
source 170
```

displays the line you want.

Since the program loops back, you might want to use the **NEXT** subcommand. This causes one COBOL verb to be executed and then returns control to the TESTCOB user. Thus, at each step of the program, you can display variables, source lines, and so on. This is useful for following COBOL transfers out of sequence, such as **GO**, **CALL**, or **PERFORM**.

If you issue the subcommand:

```
list answr
```

you get a display of the structure of the group variable ANSWR, as defined in the Data Division. Since it's a group item rather than an elementary item, the values of the variables are not displayed. However, you might find this helpful in debugging the error in the program.

The breakpoints that you set up during testing are in effect throughout the session. The **LISTBRKS** subcommand lets you display breakpoints. The **OFF** subcommand lets you remove them one by one. With the **RUN** subcommand you can continue execution and ignore all the breakpoints. At any point you can use the **END** subcommand to end the debugging session and return to VM/SP.

TESTCOB offers you other useful debugging facilities. For example, you can:

- Get a system dump of the region in which your program is executing by entering **DUMP** instead of **END**.
- Take conditional debugging actions during program execution by using the **IF** and **WHEN** subcommands.
- Determine the current status of files specified in your source program by using the **LISTFILE** command.
- Trace the program execution (paragraph by paragraph and calls to other programs) by using the **TRACE** subcommand.

For more information on **TESTCOB**, see *IBM OS COBOL Interactive Debug Terminal User's Guide and Reference.*

VS FORTRAN Interactive Debug

This section describes the VS FORTRAN Interactive Debug product. If you're using the FORTRAN Interactive Debug product, skip to "FORTRAN Interactive Debug" on page 241.

You can invoke VS FORTRAN Interactive Debug by specifying an execution-time parameter when a VS FORTRAN program is executed. It is available for use in a CMS environment, with or without ISPF.

If the following conditions are met, you can use VS FORTRAN Interactive Debug to debug a program:

- The program was compiled with a release of VS FORTRAN prior to Release 3.0 and you specified the TEST option at compile time.
- The program was compiled with VS FORTRAN Release 3.0 or later and you didn't specify the NOSDUMP option at compile-time unless you also specified TEST.

In either case, Release 3.1 or later of the VS FORTRAN Library must be made available by using the GLOBAL TXTLIB command when the compiler-produced TEXT file is LOADed or LKEDed.

After the program has been compiled, you may execute the program in line mode (without ISPF) or in full-screen mode (with ISPF).

Using VS FORTRAN Interactive Debug in Line Mode

Suppose you compiled the following program (the same one you entered in "Creating a FORTRAN File" on page 24) in accordance with the discussion above.

```
      PROGRAM MYPROG
      CHARACTER*8 F,S
      WRITE (6,5)
      READ (5,2) F
      WRITE (6,10)
      READ (5,2) S
      WRITE (6,15) F,S
2     FORMAT (A8)
5     FORMAT (' ENTER YOUR FIRST NAME.')
10    FORMAT (' AND NOW YOUR LAST NAME.')
15    FORMAT (' WELCOME TO CMS, ',A8,1X,A8)
      STOP
      END
```

If you execute this program in line mode with the DEBUG execution-time option, Interactive Debug suspends execution before beginning the program. This lets you set breakpoints or issue other Interactive Debug commands. When you respond with a **GO** command, execution begins.

Whenever execution is suspended and VS FORTRAN Interactive Debug is waiting for you to enter a command, you see the following prompt:

```
    FORTIAD
```

Suppose you want to stop the program at statement number 7, to look at the variables which are about to be written. Use the **AT** command to specify breakpoints. You can now specify a list of commands to be executed when the breakpoint is encountered. For example, the command:

```
    at 7 (list f%list s%go)
```

causes this to happen: whenever statement 7 is reached, the values of F and S are listed and execution immediately resumes (without giving you a chance to enter further commands from the terminal).

When you issue the **GO** command, the program executes and you'll see the following display (lower case indicates your response):

```
FFT06F001  ENTER YOUR FIRST NAME
FFT05F001 INPUT: PRECEDE INPUT WITH % OR ENTER IAD COMMAND
%bill
FFT06F001  AND NOW YOUR LAST NAME
FFT05F001 INPUT: PRECEDE INPUT WITH % OR ENTER IAD COMMAND
%smith
AT:  7 IN SAMPLE
F =  BILL
S =  SMITH
FT06F001   WELCOME TO CMS,  BILL     SMITH
PROGRAM TERMINATED, RC= 0
```

At this point, the program has finished executing. You can still issue some Interactive Debug commands, but you cannot restart the program. You have to issue the **QUIT** command when you want to end the debugging session.

It is good practice to establish a breakpoint before the end of the program (perhaps on the STOP statement). Then, if you want to change some variable values and return to a statement in the program, you can do so.

Using VS FORTRAN Interactive Debug in Full-Screen Mode

The same operations as shown in the line mode example are valid in the full-screen mode. The primary difference is that you're executing under ISPF, and have a more "friendly" presentation of output. A scrollable log of all input and output is provided as well as a log file of all debugging activities. See *VS FORTRAN Interactive Debug Reference* for further explanation and examples.

Other VS FORTRAN Interactive Debug Facilities

VS FORTRAN Interactive Debug offers many useful debugging facilities. For example, you can:

- Use the **ERROR** command to specify the level of remedial action to be taken, if any, when a program error is detected by the VS FORTRAN library.

- Use the **FIXUP** command to provide corrected values for the arguments that caused the error.
- Use the **IF** and **WHEN** command to take conditional debugging actions when your program is being executed.
- List the the number of times statements were executed by issuing the **LISTFREQ** command.
- Use the **TRACE** command to trace the execution of the program.
- Use the **WHERE** command to find out what statement number will be executed next, and optionally show information about how the program got there.
- Use the **SYSCMD** command to issue CMS commands without leaving Interactive Debug.
- Use commands like **ENDFILE, CLOSE, REWIND**, and **BACKSPACE** to manipulate external VS FORTRAN files.
- Use the **HELP** command to ask for information concerning any command, or about common debugging tasks.
- Use the **TERMIO** command to request that all input and output operations requested by the VS FORTRAN program are to be handled using the standard VS FORTRAN Library I/O routines rather than the special routines provided by Interactive Debug.

VS FORTRAN Version 2 Interactive Debug

VS FORTRAN Version 2 has an Interactive Debug facility as part of the product. This is very similar to VS FORTRAN Interactive Debug as described above, with some additional features:

- It supports VS FORTRAN Version 2 programs.
- It provides subroutine timing facilities to help find which parts of your program are taking the most CPU time.
- If run under ISPF V2, it can display FORTRAN source listings in a special window, without having to split the screen. Color and extended highlighting are also supported.
- It can be run in batch mode as well as interactively.

FORTRAN Interactive Debug

The FORTRAN Interactive Debug product may be used to debug FORTRAN G1 and Code and Go FORTRAN programs, as well as VS FORTRAN programs. However, you cannot successfully compile FORTRAN G1 or Load and Go programs if they contain a FORTRAN language debug packet. If the debug packet is removed, such programs become eligible for FORTRAN Interactive Debug (with VS FORTRAN programs, debug packet statements are simply ignored).

Normally, you can compile, load, and run a FORTRAN program in one step using the CMS **RUN** command. When using Interactive Debug, you need to compile the program, then load and run it. You must include the parameter TEST with the other parameters you use in the compile command for your particular version of FORTRAN.

For example, the following command compiles a FORTRAN program for debugging under CMS.

```
fortvs myvsprg (print source list map test)
```

The CSECT name of the main program is always set to MAIN, regardless of what was otherwise specified.

There are other minor differences when the TEST parameter is used. For more details, see *FORTRAN Interactive Debug for CMS and TSO Guide and Reference.*

After you've compiled your program, you'll have a stored, test-compiled TEXT file ready for processing. Before issuing the **TESTFORT** command, you must make sure that the library with the FORTRAN Interactive Debug routines is made available. This library has the default name TFORTLIB and should precede the other libraries normally specified in the **GLOBAL** command. Since **TESTFORT** needs to communicate with the terminal, you should also specify the communications routines library (default name TSOLIB).

For example, you might enter:

```
global txtlib tfortlib tsolib vfortlib cmslib fortmod2 mod2lib
```

if your program required the FORTRAN Mod II libraries FORTMOD2 and MOD2LIB. The VFORTLIB and CMSLIB are required to run VS FORTRAN.

The **GLOBAL** command remains in effect for the rest of the session. If you run a program under direct system control (instead of TESTFORT control), issue a new **GLOBAL** command without TFORTLIB or TSOLIB.

You can now issue the **TESTFORT** command. It's like **EDIT** in the way that it lets you enter subcommands to manipulate data and monitor the execution of your program.

TESTFORT provides several subcommands that let you control the execution of your program:

GO	starts the execution of the program. Until you actually issue the **GO** subcommand, TESTFORT waits. If you want to return to TESTFORT (rather than continuing with the execution of the program), press ATTN twice. The system displays **TESTFORT** to indicate that you can enter subcommands.
AT	specifies a line number where program execution is to stop. This is called a breakpoint. You can specify as many breakpoints as you like. When you specify **AT**, you can also specify variables whose values are to be displayed, as well as other subcommands (such as **GO**).
LISTBRKS	provides a list of breakpoints.
OFF	turns off a breakpoint.
LIST	specifies a variable whose value you want to be displayed.
SET	assigns the value of a program variable.
NEXT	steps through the source program one line at a time.
RUN	functions like **GO**, but ignores breakpoints it meets.
SOURCE	displays a line of the source program.
FIXUP	alters the value of a variable which has caused an execution error, and then resumes the execution of the program.
END	ends the execution of TESTFORT. (Pressing the ATTN key twice does the same thing.)

The only required parameter for TESTFORT in CMS is the name of the program you're debugging. You can use the keywords DISK, PRINT, and NOPRINT to reroute the output (which is normally at your terminal). TESTFORT locates two files from your filename: the TEXT file produced by the compilation, and the original source program file (filetype FORTRAN).

Suppose you compiled the TESTPROG program (the same one you entered in "Creating a FORTRAN File" on page 24), you would receive a listing similar to the following:

Example

```
010         PROGRAM MYPROG
020         CHARACTER*8 F,S
030         WRITE (6,5)
040         READ (5,2) F
050         WRITE (6,10)
060         READ (5,2) S
070         WRITE (6,15) F,S
080  2      FORMAT (A8)
090  5      FORMAT (' ENTER YOUR FIRST NAME.')
100  10     FORMAT (' AND NOW YOUR LAST NAME.')
110  15     FORMAT (' WELCOME TO CMS, ',A8,1X,A8)
120         STOP
130         END
```

Now you issue this command:

```
testfort testprog (disk debugfl)
```

where:

TESTPROG is the name of the textfile resulting from the FORTVS compilation.

DISK indicates that the debug output goes to a disk file.

DEBUGFL is the name of the debug disk file.

After you enter this command, the system displays **TESTFORT**. This means that the system is ready to accept a command from you. Now you can. for example, say that you want to see the values of F and S at line 70. Once you set this, you can start your program executing. The commands go like this:

```
TESTFORT
at 70 (list f;list s;go)
TESTFORT
go
```

When the program execution reaches line 70, this is displayed:

```
AT:70 IN MAIN
F = JOHN
S = SMITH
WELCOME TO CMS, JOHN SMITH
PROGRAM HAS FINISHED EXECUTION NORMALLY
TESTFORT
```

After displaying the line number, TESTFORT executes each subcommand in turn. In this example, execution of the program resumed as the **GO** subcommand was included. If **GO** is omitted, Interactive Debug displays **TESTFORT** again, and a reply of **GO** lets the session continue.

However, if you want to end the debugging session, reply with **END** to the last TESTFORT message.

In the TESTFORT environment the **SOURCE** subcommand lets you view a program statement. For example:

```
source 50
```

causes the display of line 50 of the program.

You can just as easily use **SOURCE** to display a line by preceding the FORTRAN statement number with a slash:

```
source 90
```

and

```
source /5
```

both result in a display of the same line.

When you reach line 70 again, you might want to change the value of a program variable. You can do this, as follows, with a SET subcommand:

```
TESTFORT
set s = 'jones'
TESTFORT
run
WELCOME TO CMS, JOHN JONES
```

If you receive control because of an error in execution, use the **FIXUP** subcommand to make a correction and resume the execution of the program. For example, if you had the following in your test program:

```
      .
      .
      .
        X = -46
        A = SQRT(X)
      .
      .
      .
```

you're returned to TESTFORT. This is because the argument of SQRT must be a positive number. You can change the argument of the mathematical function and continue processing. Thus:

```
fixup arg1(64)
```

applies the value 64 (instead of -46) to the square root function. The change is made only for the purposes of this function. The value of variable X remains -46, although the value of A becomes 8 (square root of 64).

TESTFORT offers many other useful debugging facilities. For example, you can:

- Specify the level of automatic remedial action to be taken, if any, when a program error is found by using the **ERROR** subcommand.

- Use the **IF** and **WHEN** subcommands to take conditional debugging actions when your program is being executed.

- List the number of times statements were executed (or were not executed) by issuing the **LISTFREQ** subcommand.

- Trace the execution of the program (branch by branch and including calls to subroutines) by using the **TRACE** subcommand.

- Get a trace of events that occurred before the last attention interrupt by using the **WHERE** subcommand.

For more information on the **TESTFORT** command, see *FORTRAN Interactive Debug for CMS and TSO Guide and Reference.*

Dialog Testing Using ISPF

If you're not using ISPF, skip to "Data Base Testing Using SQL/DS" on page 250.

Four testing modes of ISPF provide processing actions to help you debug a dialog. You can specify only one of these keywords on the **ISPSTART** command: TEST, TESTX, TRACEX, or TRACE. This controls the operational mode of ISPF during dialog testing.

Here's how the testing modes of ISPF differ from the normal mode:

- Panel and message definitions are refetched from the libraries each time you specify one in an ISPF service. Normally, frequently used panels and messages are retained in virtual storage and fetched from there rather than from the library. If you've modified the panel or message library, the testing mode ensures that the latest version is accessed during a test run.

- Tutorial panels are displayed with the current and previous panel names and the previous message **id** on the bottom line of the screen. This helps you identify the position of the panel in the tutorial flow. When you use the **PRINT** or **PRINT-HI** commands, the screen printouts show similar diagnostic data.

- If you make an error, you can force the dialog to continue. However, results from that point on are unpredictable. All other ISPF-detected errors, ABENDs, or program interrupts force an ABEND of ISPF.

If you issue the TRACE keyword with ISPSTART, the testing mode also writes a message to the ISPF log file. It does so whenever you invoke any ISPF service and whenever the service detects an error.

Issuing one of the extended mode keywords TESTX or TRACEX causes all messages written to the ISPF log file (including trace messages) to be displayed at the terminal as well.

For more details of ISPF testing mode, see *ISPF Dialog Management Services and Examples.* If your installation has ISPF/Program Development Facility (ISPF/PDF), you should use its dialog test option instead of the testing modes described above.

The dialog test option provides you with aids for testing ISPF dialog parts (functions, panels, variables, messages, tables, skeletons) and complete ISPF applications. For example, you can:

- Invoke selection panels, command procedures, programs and shared segments
- Display panels
- Add new variables and modify variable values
- Display a table's structure and status
- Display, add, modify, and delete table rows
- Browse the ISPF log
- Execute dialog services
- Add, modify, and delete function and variable trace definitions
- Add, modify, and delete breakpoint definitions.

When you enter dialog test, you enter a new user application with an application ID of ISR. All the options operate in this context.

Dialog test is itself a dialog and, therefore, uses the dialog variables. Since it is important to allow your dialog to operate without interference (as though in a production environment), dialog test accesses and updates variables independently of your dialog variables.

If your dialog encounters a severe error when it invokes a dialog service, that error is handled as requested by a dialog. The current CONTROL service ERRORS setting (CANCEL, or RETURN; the default is CANCEL) determines what is done. If CANCEL is in effect, when the error message panel is displayed you may choose whether to continue dialog testing.

We'll now discuss several of these options.

The **functions option** lets you test a dialog function (panel, command procedure, or program). You don't have to write supporting code or panels. The name of the dialog function and the parameters that may be passed are the same as those that you can specify (from a dialog function) when you invoke the SELECT service. When you press the ENTER key, a SELECT is done. When you select this option, a panel is displayed that lets you identify the dialog function that you want to test.

During panel development, the **panels option** lets you test newly created or modified panels and messages. You don't need to write supporting code to display them. Any variables referenced and set during panel processing are handled according to standard ISPF protocol.

The **variables option** lets you:

- Display all ISPF variables defined in the dialog application you're testing.
- Change the value of a variable.
- Define new variables.

When you select this option, a scrollable display indicates all the current variables for the dialog being tested. The rows of the display are ordered by the pool containing the variables, then alphabetically by variable name within each pool. The function variable pool is listed first, followed by the shared variable pool, and then the profile variable pool. Insertions are left where they are entered on the display.

Modifications to the display are processed when you press the ENTER key. Updating of the variable pools occurs when you enter the **END** command.

You can create new dialog variables, but you can't create two variables with the same name in the variable pool. You can't delete a variable, but you can set its value to nulls.

The **tables option** lets you:

- Display the contents of an existing row in an open table.
- Remove an existing row from an open table.
- Change the contents of an existing row of an open table.
- Add a new row after a selected row of an open table.
- Display the structure of a table.
- Display a data information panel reflecting all operations using a specified table.

The **log option** lets you display and browse data recorded in the ISPF log. You can use all the browse command, except BROWSE, while looking at the ISPF log. The ISPF log contains the following types of trace output:

- Trace header entries.
- Function trace entries.
- Variable trace entries.

The **dialog services option** lets you execute a dialog service by entering the service command invocation with or without the ISPEXEC characters. You can call any dialog service that is valid in the command environment except CONTROL at a breakpoint or before invoking a function.

The **traces option** lets you define, change, and delete trace specifications. You can trace executed dialog services, except for the VPUT service issued to a panel, and referenced dialog variables during dialog execution. Trace data is placed in the transaction log. From here you can browse it (using the LOG option), or print it when you exit from ISPF.

Since tracing may degrade dialog performance and create large amounts of output, care should be taken in setting the scope of trace definitions.

When you select this option, you're shown a selection panel on which you can indicate the type of trace (function or variable) you wish to define.

Use the **function trace option** to establish criteria for recording the names of dialog service calls, the service parameters, and return code in the ISPF log. Service calls made by the dialog or during test processing are recorded. Whenever a new application or function has data recorded, a header is placed in the trace. When you select the function trace option, a scrollable panel displays all currently defined function traces. You may add, delete, and modify function trace definitions using this panel before invoking a function.

The **variable trace option** is used to establish criteria for recording variable usage. The usage of a variable is recorded:

- If an ISPF service is directly asked to operate on the variable (for example, VGET, VPUT, VCOPY).

- If an ISPF service is indirectly asked to operate on the variable (for example, DISPLAY).

Variables changed under the variables option are also recorded if the trace specifications are met.

When you select the variable trace option, a scrollable display lists all currently defined variable traces. You may add, delete, or modify variable trace definitions by using this panel before invoking a function.

A **breakpoint** is a location at which the execution of a dialog is suspended so that dialog test facilities may be used. The breakpoint option lets you indicate where such temporary suspensions should occur. At a breakpoint, you're given control. You may now examine and manipulate dialog data (tables, variables, etc.) using various test options. You can also specify new test options, such as traces and other breakpoints.

Breakpoints are located immediately before a dialog service receives control or after it relinquishes control. Breakpoint definitions cause special handling within the ISPLINK and ISPEXEC interfaces to dialog services. No user dialog is modified.

When you select the breakpoint option, a scrollable display shows all currently defined breakpoints for this session. You may add, delete, or modify breakpoint definitions using this panel before invoking a function or a breakpoint. All breakpoints exist until you delete them or you end or cancel your dialog test session. If you invoke a dialog function or a selection panel and encounter a breakpoint, the dialog test breakpoint primary option menu is displayed.

Like the dialog test primary option menu, the breakpoint primary option menu lets you use the **RETURN** command from any one of the selected test options to process a redisplay of the breakpoint primary option menu. You must use:

- The GO option to terminate processing at this breakpoint and continue executing the dialog being tested.

- The CANCEL option to cancel the dialog test option. This protects against inadvertent loss of data.

The breakpoint primary option menu contains all options of the dialog test primary options menu. It therefore presents all of the dialog test functions to you.

When a user dialog encounters a breakpoint, the current dialog environment is saved. When you select the GO option, the environment is restored, except for the following:

- If you change variable, table, and file tailoring data at a breakpoint, these actions are performed as an extension of the suspended dialog. It is as if the dialog takes all the actions itself during execution.

- If you modify the service return code (on the breakpoint primary option menu), the new return code is passed back to the dialog. It is as if the service sets the new code itself.

- If you execute the **PANELID** command at the breakpoint, the last setting for displaying panel identifiers is retained.

- If any CONTROL service settings for DISPLAY LINE or DISPLAY SM are in effect before the breakpoint, such settings are lost.

The manipulation of one dialog part may cause a change to another dialog part.

The dialog test option also lets you manipulate a table, to display its structure and status, and to browse the ISPF log.

For further information on these functions and all dialog test functions see *ISPF/PDF for VM/SP Program Reference.*

Data Base Testing Using SQL/DS

If you're not using SQL/DS, skip to "Using CMS Debugging Facilities" on page 252.

You can use SQL/DS as a tool for prototyping data designs and implementations during the application development process. For example, the ability to CREATE, ALTER, and DROP tables dynamically from an online, interactive environment lets you experiment with different designs.

SQL/DS facilities support these data prototyping functions:

- Online definition of model designs.
- Generation/loading of test data.
- Design documentation and analysis.

You can use ISQL to enter table, view, and index definitions for validating and testing data design. The interactive definition through ISQL offers you direct feedback on definitional errors. This feedback addresses both syntax and data mapping errors.

If you enter SQL definitional commands using ISQL, then it can save them as stored queries for later recall, modification, or rerun. You can also save statements in CMS files used as input (SYSIN) to the DBS utility.

You can load tables created for design purposes with test data using these SQL/DS facilities:

- Item by item, using the ISQL **INPUT** command.
- From existing SQL/DS tables within the data base, using the SQL **INSERT** command.
- From existing SQL/DS tables in another data base, using the DBS **UNLOAD** and **RELOAD** commands.

For more information on this topic, see *SQL/DS Planning and Administration — VM/SP*.

By using the SQL/DS explanation tables and **EXPLAIN** command, you can analyze how a given design will perform. You can issue the **EXPLAIN** command via ISQL, the DBS utility, or an application program. **EXPLAIN** lets you get information about the structure and execution performance of a SQL command.

You can see how well a **SELECT** command performs by using the ISQL query cost estimate. ISQL displays this at the end of every SELECT result. This estimate of the resources used during command execution is related to, but isn't the same as, that obtained via **EXPLAIN**.

For full details on how SQL/DS performs, see *SQL/DS Planning and Administration — VM/SP.*

You can use ISQL facilities to test and debug SQL commands for application development. The ISQL support of routines lets you develop logical sequences of SQL commands for this purpose. You can produce different routines using parameters to simulate program variables for various paths through the application logic. This tests the functional results of an application against various inputs.

In these situations you can use the ISQL command **SET RUNMODE**, which lets you stop or continue the execution of an ISQL routine when an error occurs.

This command offers these options:

- Continue to the next command even if an error occurs. (You can use this option to bypass unconnected errors or examine later ones.)

- Stop processing when you make an error. But don't perform ROLLBACK WORK (that is, leave the data in its processed state).

- Stop processing when you make an error. But perform ROLLBACK WORK (that is, erase all changes the routine made and preserve the integrity of the data base).

For more details on this subject, see *SQL/DS Terminal User's Guide — VM/SP.*

When you're developing programs, you may want to use the SQL **INCLUDE** command. This is useful when many applications use the same host variables or SQL command sequence. This command causes the preprocessors to include source lines from other CMS files in your source code. For example, you might place a lengthy **SELECT** command in a separate CMS file and use it in various programs by coding **INCLUDE** commands.

For example, in a COBOL or FORTRAN program, you'd do this by coding the filename of the CMS file:

```
EXEC SQL INCLUDE SOURCE1 END-EXEC.
```

at the point in the source code where you include the **SELECT** command.

When developing a program with embedded SQL commands, you can run the SQL/DS preprocessors with a **CHECK** option. This causes the preprocessor to generate diagnostics on the SQL in the program but not an access module or compiler input. You can thus use a skeleton of the final program to do a lot of initial code development and debugging.

If you'd like more details on the preprocessors, see *SQL/DS Application Programming.*

Using CMS Debugging Facilities

CMS provides a number of commands that are useful in debugging programs. These include DEBUG, GO, BREAK, COMPARE, SET, STORE, SVCTRACE, and PER. We'll discuss each of these below.

Using the DEBUG Command

When you use the **DEBUG** command, you can enter the VM debug environment. If you need to debug programs at this level, you may want to have on hand the *VM/370 Principles of Operation*.

Once you enter the debug environment, VM saves the contents of all general registers, the channel status word (CSW), and the channel address word (CAW). When you leave the environment, you can examine and change the contents before restoring them. When a program ends abnormally (abends), VM checks if the next command entered is **DEBUG**. If it is, it saves the contents of the general registers, the CSW, and the CAW, plus the old program status word (PSW) from the time of the abend.

You can enter the **DEBUG** subcommands **CAW**, **CSW**, and **PSW** to display the contents of the corresponding words. **GPR n** displays the contents of general purpose register n. **X hh** displays the contents of main storage at hex address hh. (Note that the start address of the program you were running is X'20000'.)

Debug provides you with certain environment commands. For example, **RETURN** lets you return to CMS, and **HX** ends the debug session completely. You can also restart your program from a specific address with **GO xxxxx**, or set up breakpoints for reentering the debug environment with the **BREAK** command.

Note: You may use debug to complement the dialog test option of ISPF/PDF. If, however, you need to examine ISPF storage areas, you can't use DEBUG.

See *VM/SP CMS Command Reference* for full details on these and other debug environment subcommands. Also, you can find further hints on the usage of the debug environment in *VM/SP CMS User's Guide.*

Using the COMPARE Command

COMPARE is a useful command. You can use it to compare the contents of two disk files of fixed or variable-length format on a record-for-record basis. Different records are displayed on the terminal. The command has the option of restricting the comparison to specific columns. This means that you could, for example, check for differences only in a key field.

Using the SET and STORE Commands

You can alter locations by using the **SET** and **STORE** commands. The format of the SET command is one of the following:

```
SET xxx hhhhhhhh
SET GPR n hhhhhhhh
```

where:

xxx	is CAW, CSW, or PSW.
hhhhhhhh	is the data to be stored.
n	is the number of the general register.

The format of the STORE command is as follows:

```
STORE xxxxx hhhhhhhh
```

where:

xxxxx	is the main storage address to be altered in hexadecimal.
hhhhhhhh	is the data to be stored.

Using the SVCTRACE Command

SVCTRACE provides you with a record of all supervisor calls in your VM.

The information, which is routed to your printer, includes:

- Call and return address information.
- GPR and floating-point register contents before, during, and after the call.

If you use more than one printer on your VM, you may want to route the trace information to a separate printer from your program output. Depending on the type of problem, sometimes it's more informative to intermix the two outputs. See *VM/SP CP Command Reference* for a full description of SVCTRACE.

Using the PER Command

You can use the **PER** command for monitoring the following lower level events during execution:

- Fetching and execution of a machine code instruction.
- Execution of successful branch instructions.

- Alterations of a specific general purpose register.
- Access to a particular area of main storage.

Here are some options that **PER** provides:

- Routing trace information to the terminal or printer.
- Reporting only every nth event monitored.
- Letting you step through the code instruction by instruction.

For more information about the **PER** instruction, see *VM/SP CP Command Reference for General Users.*

Summary

VM and CMS provide a number of testing and debugging facilities. Interactive Debug contains a command and a set of subcommands that aid you in diagnosing and solving problems in your programs. You can use COBOL Interactive Debug under CMS to debug any COBOL program compiled with the TEST option. You can invoke VS FORTRAN Interactive Debug by specifying an execution-time parameter when a VS FORTRAN program is executed. You can also use the FORTRAN Interactive Debug for VS FORTRAN programs. There are four testing modes of ISPF provide processing actions to help you debug a dialog. You can use SQL/DS as a tool for prototyping data designs and implementations during the application development process. You can use ISQL facilities to test and debug SQL commands for application development.

When you use the **DEBUG** command, you can enter the VM debug environment. Once you enter the debug environment, VM saves the contents of all general registers, the channel status word (CSW), and the channel address word (CAW). When you leave the environment, you can examine and change the contents before restoring them.

Chapter 11. Using the VM/SP HPO Vector Facility Support with FORTRAN Programs

This chapter describes the VM/SP HPO Vector Facility support and how to use the associated commands. The following topics are discussed:

- A brief overview of the Vector Facility is given with references to documentation where additional information on the Vector Facility can be found.
- How to display the Vector Facility registers at your terminal.
- How to display at your terminal other Vector Facility values such as the vector activity count, vector status register, and vector mask register.
- How to change the contents of the Vector Facility registers and values.
- Error messages that you may receive using the Vector Facility commands.
- How to display how much Vector Facility resources your virtual machine is using.
- A brief discussion on vector instruction tracing.

Vector Facility Overview

VM/SP HPO supports the Vector Facility in System/370 mode. The Vector Facility is an instruction processor that can manipulate values at a high speed, usually floating-point values.

The Vector Facility consists of:

- Additional registers:
  - 16 vector registers, each of which contains a number of 32-bit elements.
  - A vector mask register, which is used by various vector instructions.
  - A vector status register, which contains information that describes the current status of the Vector Facility.

  - A vector activity count, which provides a means of measuring the time required to execute instructions of the Vector Facility.

- 171 vector instructions

For more information about the Vector Facility, refer to *System/370 Vector Operations*.

To execute programs the contain vector instructions, VM/SP HPO must be run on a processor with a real Vector Facility. CP does not simulate a Vector Facility when an actual Vector Facility is not configured.

If a Vector Facility is available on your system, your compiled FORTRAN object programs can use its array processing capabilities. Note that the Vector Facility is supported by the VS FORTRAN Version 2 program. For specific options and other additional information on this support, refer to *VS FORTRAN Application Programming Guide*.

Your use of the Vector Facility starts when either your program issues a vector instruction or you enter a DISPLAY or STORE command associated with the Vector Facility such as DISPLAY VR (display vector register). You can use the DISPLAY and STORE commands to display and change the various register sets in the Vector Facility.

If you receive the following message,

```
VECTOR FACILITY NOT AVAILABLE
Ready;
```

ask the system operator to issue the VARY ONLINE VECTOR command.

Note: You would also get this message if no Vector Facility is on your system.

If your program executes a vector instruction and the Vector Facility is not available, it will receive a program check.

Your use of the Vector Facility ends when:

- You enter the LOGOFF command

- You enter the SYSTEM CLEAR command. (Note that if you issue this command, you will have to IPL again.)

- The system operator issues a FORCE command, forcing your virtual machine off the system, or

- You enter IPL xxx CLEAR or IPL a named system (for example, CMS).

When your use of the Vector Facility ends, CP releases the vector register save areas for your virtual machine.

Displaying Vector Facility Registers

You can use the DISPLAY command to display the contents of your virtual machine's vector registers. To do this, set a breakpoint at the place in your program where you want to test the Vector Facility's registers. (See "Chapter 10" for information on how to set breakpoints.) When execution of your program stops, enter the DISPLAY command to view the contents of the registers.

Note: The first time a vector instruction is executed in an application, the vector environment has been disabled and a vector operation exception occurs. CMS enables the vector environment, clears the vector status register and then reissues the instruction. If you have set a BREAK point at this instruction, the contents of the vector status register you display may not represent what your program will see.

To determine whether CMS will clear the vector status register when it encounters this instruction, display control register 0. If the vector control bit is on (CR0, bit 14, 00020000 bit), CMS has already determined that this is a vector application and the clearing will not occur.

Note: CMS resets the Vector Status Register between commands. EXEC2 and System Product Interpreter (REXX) users must do this themselves by issuing the EXECOS command (OS and VSAM reset).

Examples

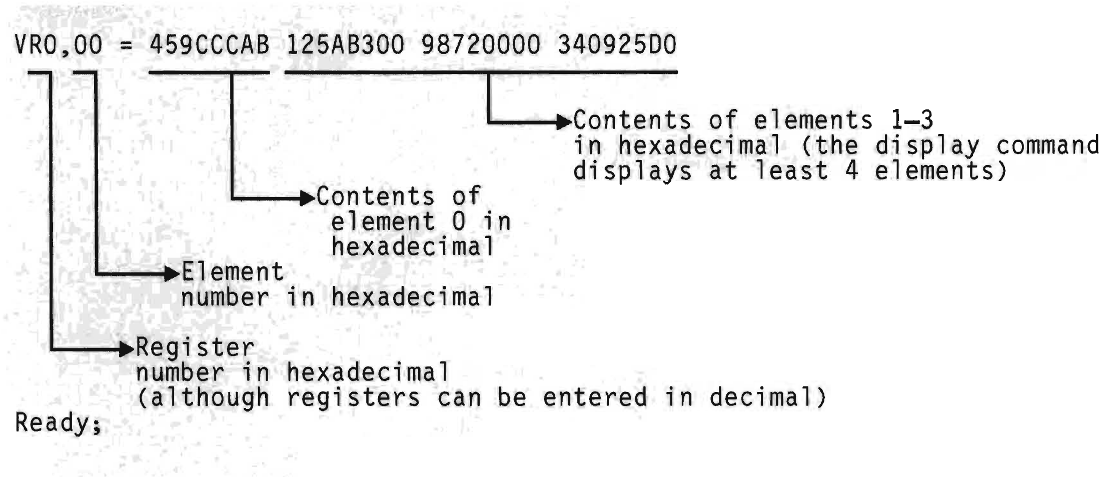
The following examples show the format of messages the system will display in response to different variations of the DISPLAY command.

Displaying the Contents of a Vector Register

To display the contents of vector register 0, enter:

```
d vr0
```

Since, in this example, you did not specify an element, the system assumes element 0. Your screen should look like this:



```
VR0,00 = 459CCCAB 125AB300 98720000 340925D0
                                    ►Contents of elements 1-3
                                     in hexadecimal (the display command
                                     displays at least 4 elements)
                     ►Contents of
                       element 0 in
                       hexadecimal
           ►Element
           number in hexadecimal
        ►Register
        number in hexadecimal
        (although registers can be entered in decimal)
Ready;
```

If you did not use the Vector Facility or your program did not use vector register 0, your screen will look like this:

```
VR0,00 = ZEROS (IN-USE BIT OFF)
```

Note: *The in-use bits, when on, show that the register has been used previously. When off, they mean no vector instructions or STORE VR commands have changed that register.*

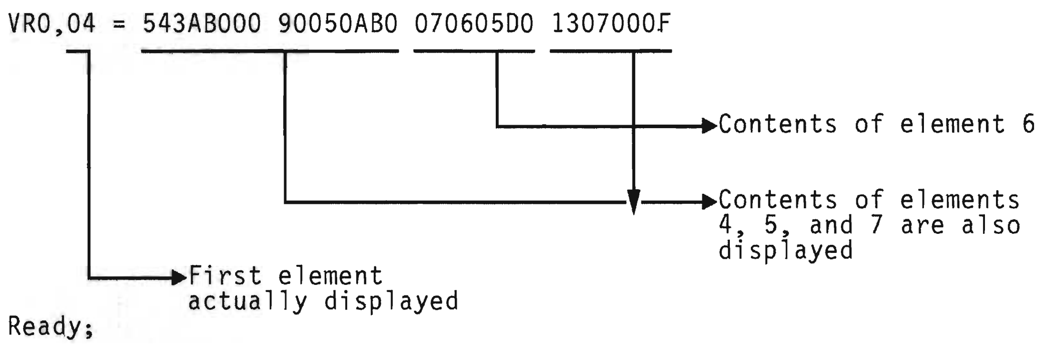
Displaying the Contents of a Specific Element

To display the contents of vector register 0 element 6, enter:

`d vr0,6 or d vr,6` (register 0 is the default)

Note: *Elements must be specified in hexadecimal. If this were element 24 decimal, for example, you would have to specify it as 18.*

Your screen should look now like this:



```
VR0,04 = 543AB000 90050AB0 070605D0 1307000F
Ready;
```

Contents of element 6
Contents of elements 4, 5, and 7 are also displayed
First element actually displayed

Displaying the Contents of a Range of Elements

To display vector register 0 elements 2 to 10 (hexadecimal A), enter:

`d vr0,2-A or d vr,2-A`

Your screen should now look like this:

```
VR0,00 = 45689001 ABC10000 0F020000 0F030000
VR0,04 = 045400C0 0C050000 09060035 0A070000
VR0,08 = 0A080000 0529000F 040B0003 050D0FFF
Ready;
```

Note that elements 0, 1, and 11 are also displayed.

Displaying the Contents of All Elements in a Range of Registers

To display the contents of all elements in registers 0 to 15 (F hexadecimal), enter:

```
d vr0-F,0-end
```

Note: *The END operand shows that you want to look at all of the elements in the specified registers starting with the first element specified. Our example says display all elements from 0 to the end of the elements. You can also use the END operand to specify all registers after the first one specified (for example, d vr0-end).*

Your screen should now look like this (the dots represent data that is displayed but not shown in this book):

```
VR0,00 = 00000000 0F010000 08920003 0F030000
VR0,04 = 0504000C 05050000 0D060002 03070000
    .          .        .        .        .
VR0,7C = 0F7C0000 067D0000 007E000F 037F0400
VR1,00 = 13000000 13010000 13020000 13030000
    .          .        .        .        .
VRF,7C = FD7C000E FE7D0000 F07E0002 F47F0200
Ready;
```

Note: *Since 128 elements are shown, the last line starts with element 124 (7C hexadecimal).*

Displaying the Contents of a Number of Consecutive Elements

To display the contents of six consecutive elements starting from element 19 (13 hexadecimal) in vector registers 3 to 5, enter:

```
d vr3-5,13.6
```

Note: *The dot (.) specifies the number of elements including element 19 (13 hexadecimal) that are to be displayed (that is elements 19 (13), 20 (14), 21(15), 22(16), 23(17), and 24(18) are to be displayed).*

Your screen should now look like this:

```
VR3,10= 3F1300E0 FC140333 3D153456 3616FFFF
VR3,14= 00170030 30180DD0 3E190400 38180091
VR3,18= 00170030 30180DD0 3E190400 38180091
VR4,10= 90000000 00000000 00000000 00000000
VR4,14= 4F130000 4E140000 43150000 49160020
VR4,18= 46170070 4A180000 4A180000 4A1A0FF0
VR5,10= 04130050 55140030 551500AA BB160004
VR5,14= 12907586 481800AB 5F190989 59989429
VR5,18= 12907586 481800AB 5F190989 59989429
Ready;
```

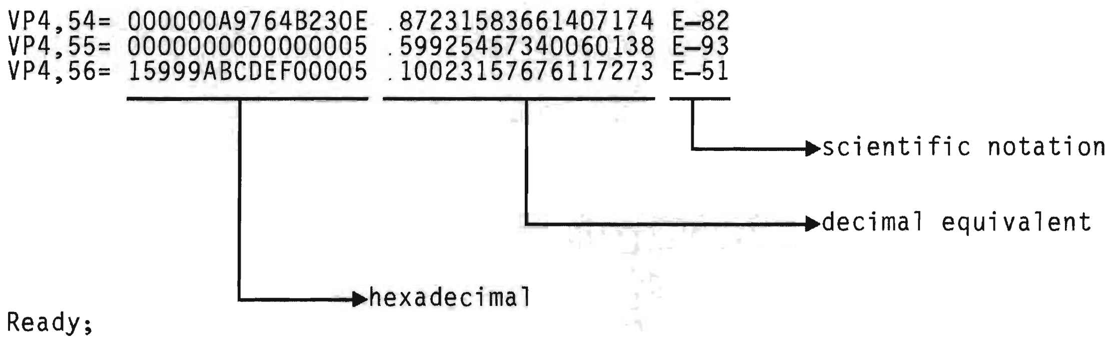
Note: *The system displays twelve elements although only six were requested; element 19 (13 hexadecimal) is the fourth element in the first, fourth, and seventh lines while element 24 (18 hexadecimal) is the first element in lines three, six, and nine.*

Displaying the Contents of a Vector Register Pair

To display the contents of vector register pair 4 and 5 (register pairs are always even/odd) elements 84 to 86 (54 to 56 hexadecimal), enter:

```
d vp4,54-56
```

Your screen should now look like this:



```
VP4,54= 000000A9764B230E .87231583661407174 E-82
VP4,55= 0000000000000005 .59925457340060138 E-93
VP4,56= 15999ABCDEF00005 .10023157676117273 E-51
Ready;
```

Note: The register pair is displayed in hexadecimal followed by the decimal equivalent in scientific notation (E-82 means 10 to the minus 82nd power). (The contents of the elements are considered to be in floating-point format.)

Displaying the Contents of a Range of Vector Register Pairs

To display the contents of element 84 (54 hexadecimal) in three register pairs starting with register pair 0 and 1, enter:

```
d vp0.3,54
```

Your screen should now look like this:

```
VP0,54 = F0000000000689A5 -.37323520244320472 E 47
VP2,54 = 8000000000000019 -.29962728670030069 E-92
VP4,54 = F000000000001000 -.35681192317648997 E 45
Ready;
```

Displaying the Contents of a Range of Elements in a Number of Consecutive Vector Register Pairs

To display the contents of elements 84 to 86 (54 to 56 hexadecimal) in three register pairs starting with register pair 0 and 1, enter:

```
d vp0.3,54-56
```

Your screen should now look like this:

```
VP0,54 = 0419269319908698 .55604615576442853 E-73
VP0,55 = F419269319908698 -.40415587850529355 E 62
VP0,56 = FF0000000000000A .10043362776618689 E 61
VP2,54 = E000000001000000 .79228162514264337 E 29
VP2,55 = 0000000000000001 .11985091468012027 E-93
VP2,56 = A000000000000345 .34135498998217281 E-52
VP4,54 = D000000000000008 .20479999999999999 E-04
VP4,55 = FFFFFFFFFFFFFFFF .72370055773322621 E 76
VP4,56 = 0000000000000001 .11985091468012027 E-93
Ready;
```

Displaying the Vector Activity Count, Vector Status Register, and Vector Mask Register

You can also display your virtual machine's vector activity count, vector status register, and vector mask register. To do this, set a breakpoint at the place in your program where you want to test one of these. (See "Chapter 10" for information on how to set breakpoints.) When execution of your program stops, enter the DISPLAY command to view the contents of the item.

Examples

Displaying the Vector Activity Count

To display the vector activity count, enter:

```
d vac
```

Your screen should now look like this:

```
VAC = 00000004 00000000
Ready;
```

Displaying the Contents of the Vector Mask Register

To display the contents of the vector mask register, enter:

```
d vmr
```

Your screen should now look like this (assuming 128 elements):

```
VMR = FFFFFFFF FFFF0000 00000000 00000000
Ready;
```

Note that, in the example, the mask bits for elements 0 through 47 are on.

Displaying the Contents of the Vector Status Register

To display the contents of the vector status register, enter:

```
d vsr
```

Your screen should now look like this:

```
VSR = 00000001 0000FFFF
Ready;
```

For the meaning of the fields in the vector status register, refer to *System/370 Vector Operations.*

Changing the Contents of Your Vector Facility's Registers

You can use the STORE command to change the contents of your virtual machine's vector registers, vector activity count, vector status register, and vector mask register. To do this, set a breakpoint at the place in your program where you want to change the Vector Facility's registers. (See "Chapter 10" for information on how to set breakpoints.) When execution of your program stops, enter the STORE command to change the contents of the registers.

Note: *The STORE command is different from the DISPLAY command in that you can only store in one register or one register pair in a single invocation of the command.*

Note: *If you are running in CMS, refer to the note under "Displaying Vector Facility Registers." If the vector facility has not been used previously, the values you store may be changed.*

Examples

Storing into a Specific Element

For example, to store hexadecimal FEDCBA98 in element 32 (20 hexadecimal) of register 3, enter:

```
st vr3,20 FEDCBA98
```

Note: *The STORE command will accept less than eight digits as a data word. For example, you could enter st vr3,20 FEDCBA9. Here, the seven digits you enter would be placed in the* **rightmost** *seven bytes of element 20 (hex) of register 3 and the leftmost byte would be set to 0. Hence, the element's contents would be changed to 0FEDCBA9. However, if you enter more than eight digits without an intervening blank (for example, st vr3,20 FEDCBA987), you will receive an error message.*

When the store operation is complete, your screen will look like this:

```
ST VR3,20 FEDCBA98
STORE COMPLETE
Ready;
```

Storing into a Number of Consecutive Elements

For example, to store hexadecimal FEDCBA98 in elements 32 (20 hexadecimal) and the next three elements of register 3, enter:

```
st vr3,20 FEDCBA98 FEDCBA98 FEDCBA98 FEDCBA98
```

When the store is complete, your screen will look like this:

```
ST VR3,20 FEDCBA98 FEDCBA98 FEDCBA98 FEDCBA98
STORE COMPLETE
Ready;
```

Storing into an Element of a Vector Register Pair

To store hexadecimal FEDCBA98FEDCBA98 in element 32 (20 hexadecimal) of register pair 2,3; enter:

```
st vp2,20 FEDCBA98FEDCBA98
```

Note: The STORE command will accept less than 16 digits as a data word. For example, you could enter st vp2,20 FEDCBA9. Here, the seven digits you enter would be placed in the **leftmost** *seven bytes of element 20 (hex) of register pair 2 and 3 and the remainder of the doubleword would be filled with zeroes. Hence, the element's contents would be changed to FEDCBA9000000000. However, if you enter more than sixteen digits without an intervening blank (for example, st vp2,20 FEDCBA98712345678), you will receive an error message.*

When the store is complete, your screen will look like this:

```
ST VP2,20 FEDCBA98FEDCBA98
STORE COMPLETE
Ready;
```

Storing into the Vector Status Register

To store 00200080 0080FFFF in the vector status register, enter:

```
st vsr 00200080 0080FFFF
```

When the store is complete, your screen will look like this:

```
ST VSR 00200080 0080FFFF
STORE COMPLETE
Ready;
```

Note: *Refer to System/370 Vector Operations for the meaning and format of the vector status register fields.*

Storing into the Vector Activity Count

To store FEDCBA98 in the first fullword of the vector activity count, enter:

```
st vac FEDCBA98
```

When the store is complete, your screen will look like this:

```
ST VAC FEDCBA98
STORE COMPLETE
Ready;
```

Storing into the Vector Mask Register

To store FFFF0000 in the first fullword of the vector mask register, enter:

```
st vmr FEDCBA98
```

When the store is complete, your screen will look like this:

```
ST VMR FEDCBA98
STORE COMPLETE
Ready;
```

Error Messages

If you enter one of the DISPLAY or STORE command vector subcommands and:

- Your virtual machine has no vector register save area and CP cannot create one because the Vector Facility is not in the configuration or offline, the system displays the message:

```
VECTOR FACILITY NOT AVAILABLE
Ready;
```

- You specified an invalid element, the system displays the message:

```
INVALID ELEMENT - nn
Ready;
```

- You specified an invalid register or element range, the system displays the message:

```
INVALID RANGE - nn-nn
Ready;
```

- If you specified an invalid register, the system responds:

```
INVALID REGISTER - n
Ready;
```

- If you specified an invalid operand, the system responds:

```
INVALID OPTION - option
Ready;
```

- If you specified invalid *hexdata* in the STORE command, the system responds:

```
INVALID HEXDATA - hexdata
Ready;
```

Displaying How Much Vector Facility Resource Your Virtual Machine is Using

You can use the INDICATE USER command to display, along with other system values, how much Vector Facility resource your virtual machine is using. If your virtual machine has a virtual Vector Facility and you enter the INDICATE USER command, besides the usual system values, the system displays:

- The total time your virtual machine used a Vector Facility since the last logon or ACCOUNT command (VVECTIME), and
- The total time your virtual machine used a Vector Facility plus the time CP has used the Vector Facility for your virtual machine since the last logon or ACCOUNT command (TVECTIME).

For the meaning of the other fields of the INDICATE USER command, refer to *VM/SP HPO CP Command Reference for General Users.*

To use the INDICATE USER command, enter:

```
INDICATE USER
```

A sample of what the system might display is:

```
PAGES: RES-0073 WS-0073 READS=000347 WRITES=000135 PG -0000 PP -0029
VTIME=001:00 TTIME=001:50 SIO=000426 RDR-000000 PRT-000000 PCH-000000
SWAPS: SWAPOUT=000362 SWAPIN=000362 SW-0000
VVECTIME=000:12 TVECTIME=000:20
Ready;
```

where:

VVECTIME is the total time in minutes:seconds your virtual machine used a Vector Facility since the last logon (or ACNT command).

TVECTIME is the total time in minutes:seconds your virtual machine used a Vector Facility plus the time CP has used the Vector Facility for your virtual machine since the last logon (or ACNT command).

Vector Instruction Tracing

When single stepping through a program using the TRACE or PER command, the instructions return their mnemonic. Some vector instructions have two mnemonics for the same opcode, one for binary operands and one for short floating-point operands. Instructions that have two mnemonics for a single opcode return the mnemonic for the binary version of the opcode.

For additional information on testing and debugging your program and on using the TRACE and PER commands, refer to "Chapter 10. Testing and Debugging Programs under VM/SP."

Appendix A. Complete COBOL Program Examples

Simple COBOL Program

The following is the simple COBOL program used in "Chapter 2: Developing Programs Using CMS" on page 19, as well as in "Chapter 10: Testing and Debugging Programs under VM/SP" on page 233

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MYPROG.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  FNAME          PIC X(22) VALUE "ENTER YOUR FIRST NAME.".
77  LNAME          PIC X(23) VALUE "AND NOW YOUR LAST NAME.".
01  ANSWR.
    05  ANSLT      PIC X(16) VALUE "WELCOME TO CMS, ".
    05  AFRST      PIC X(8)  VALUE SPACES.
    05  FILLER     PIC X     VALUE SPACES.
    05  ALAST      PIC X(8)  VALUE SPACES.
PROCEDURE DIVISION.
    DISPLAY FNAME UPON CONSOLE.
    ACCEPT  AFRST FROM CONSOLE.
    DISPLAY LNAME UPON CONSOLE.
    ACCEPT  ALAST FROM CONSOLE.
    DISPLAY ANSWR UPON CONSOLE.
    STOP RUN.
```

Complete COBOL Program

The following COBOL program (called COBOL1) lets the user add, change, delete, or display records in a file of peoples' names, by serial number. Records must be added before they can be changed, deleted or displayed.

COBOL1 should be compiled and put in module form, created by the GENMOD command:

```
GLOBAL TXTLIB COBLIBVS
LOAD COBOL1
GENMOD COBOL1
```

EXEC for Complete COBOL Program

The following EXEC called DRIVE1, is an example of the type of procedure that you might use to drive the program given in "COBOL Program" on page 273.

```
&TRACE
STATE WORK DATA A
&IF &RETCODE GT 0 &GOTO -OK
&TYPE FILE 'WORK DATA A' EXISTS.  ERASE AND TRY AGAIN
&EXIT
-OK
FILEDEF NAMES DISK NAMES DATA
FILEDEF WORK DISK WORK DATA
COBOL1
&IF &RETCODE NE 0 &GOTO -NG
STATE WORK DATA A
&IF &RETCODE GT 0 &GOTO -NF
ERASE NAMES DATA A
RENAME WORK DATA A NAMES DATA A
-NF
&EXIT
-NG
ERASE WORK DATA A
&EXIT
```

DRIVE1 invokes the program COBOL1. It does the file management for COBOL1 using CMS commands. The program creates a temporary work file, so DRIVE1 checks if the file already exists. If so, it issues an error message, and does not call the program. Otherwise it issues the FILEDEF commands and calls COBOL1. Upon return, DRIVE1 tests the return code set by COBOL1. If the return code indicates an incomplete work file, it is erased. If the return code indicates a completed work file, the old master file is erased and the work file renamed as the new master file.

COBOL Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBOL1.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INFILE ASSIGN TO DA-3330-S-NAMES
    ACCESS MODE IS SEQUENTIAL.
    SELECT OUTFILE ASSIGN TO DA-3330-S-WORK
    ACCESS MODE IS SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD  INFILE
    RECORDING MODE IS F
    LABEL RECORDS OMITTED
    DATA RECORD IS EMPRECIN.
01  EMPRECIN.
    03   SERIALNIN   PIC X(6).
    03   FRSTNMIN    PIC X(16).
    03   LASTNMIN    PIC X(16).
FD  OUTFILE
    RECORDING MODE IS F
    LABEL RECORDS OMITTED
    DATA RECORD IS EMPRECOUT.
01  EMPRECOUT.
    03   SERIALNOUT  PIC X(6).
    03   FRSTNMOUT   PIC X(16).
    03   LASTNMOUT   PIC X(16).
WORKING-STORAGE SECTION.
01  FNAME     PIC X(16)  VALUE SPACES.
01  LNAME     PIC X(16)  VALUE SPACES.
01  INPLINE1.
    03   FNTYPE    PIC X        VALUE SPACES.
    03   FILLER    PIC X.
    03   EMPSER    PIC X(6)   VALUE SPACES.
01  ERRMSG    PIC X(20)  VALUE "INCORRECT SERIAL NO.".
01  GOODMSG   PIC X(20)  VALUE "OPERATION COMPLETED.".
01  MENULINE1 PIC X(21)  VALUE "ENTER FUNCTION NUMBER".
01  MENULINE2 PIC X(27)  VALUE "(1-ADD, 2-CHANGE, 3-ERASE, ".
01  MENULINE3 PIC X(17)  VALUE "4-DISPLAY, 5-END)".
01  MENULINE4 PIC X(21)  VALUE "& REQUIRED SERIAL NO.".
01  RECFRSTNM PIC X(18)  VALUE "ENTER FIRST NAME: ".
01  RECLASTNM PIC X(17)  VALUE "ENTER LAST NAME: ".
01  RECFLAG   PIC X      VALUE "I".
    88  REC-FOUND        VALUE "F".
    88  SKIP-REC         VALUE "S".
    88  END-OF-FILE      VALUE SPACES.
PROCEDURE DIVISION.
    DISPLAY MENULINE1 UPON CONSOLE.
    DISPLAY MENULINE2 MENULINE3 UPON CONSOLE.
    DISPLAY MENULINE4 UPON CONSOLE.
    ACCEPT INPLINE1 FROM CONSOLE.
    IF   FNTYPE > 0 AND FNTYPE < 5  THEN
    OPEN INPUT INFILE OUTPUT OUTFILE
    PERFORM FINDREC UNTIL END-OF-FILE
    CLOSE INFILE
    CLOSE OUTFILE.
STOP RUN.
```
(Continued on next page)

```
FINDREC.
    PERFORM READREC.
    IF   EMPSER = SERIALNIN THEN
         MOVE "F" TO RECFLAG
         IF   FNTYPE NOT = 1 THEN
              IF   FNTYPE = 3 THEN
                   DISPLAY GOODMSG UPON CONSOLE
                   MOVE "S" TO RECFLAG
                   PERFORM COPYREST UNTIL END-OF-FILE
              ELSE
                   MOVE FRSTNMIN TO FNAME
                   MOVE LASTNMIN TO LNAME
                   PERFORM DISPNAME
         ELSE
              DISPLAY ERRMSG UPON CONSOLE
              PERFORM COPYREST UNTIL END-OF-FILE
    ELSE
         IF   NOT END-OF-FILE THEN
              MOVE EMPRECIN TO EMPRECOUT
              PERFORM WRITEREC
         ELSE
              IF   FNTYPE = 1 THEN
                   MOVE SPACES TO FNAME
                   MOVE SPACES TO LNAME
                   PERFORM DISPNAME
              ELSE
                   DISPLAY ERRMSG UPON CONSOLE.
DISPNAME.
    DISPLAY FNAME LNAME UPON CONSOLE.
    IF   FNTYPE = 4 THEN
         DISPLAY GOODMSG UPON CONSOLE
         PERFORM COPYREST UNTIL END-OF-FILE
    ELSE
         MOVE EMPSER TO SERIALNOUT
         DISPLAY RECFRSTNM UPON CONSOLE
         ACCEPT  FRSTNMOUT FROM CONSOLE
         DISPLAY RECLASTNM UPON CONSOLE
         ACCEPT  LASTNMOUT FROM CONSOLE
         DISPLAY GOODMSG UPON CONSOLE
         PERFORM WRITEREC
         IF   FNTYPE = 2 THEN
              MOVE "S" TO RECFLAG
              PERFORM COPYREST UNTIL END-OF-FILE.
COPYREST.
    IF   SKIP-REC THEN
         MOVE "F" TO RECFLAG
    ELSE
         IF   NOT END-OF-FILE THEN
              MOVE EMPRECIN TO EMPRECOUT
              PERFORM WRITEREC.
    IF   NOT END-OF-FILE THEN
         PERFORM READREC.
READREC.
    READ INFILE AT END
         MOVE SPACES TO RECFLAG.
WRITEREC.
    WRITE EMPRECOUT.
```

Complete COBOL Program Using ISPF

The following COBOL program does the same things as the program in the previous section, but it uses ISPF.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SCOBOL2.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  EMPSER          PIC X(6) VALUE SPACES.
01  LEMPSER         PIC 9(6) VALUE 6 COMP.
01  NEMPSER         PIC X(8) VALUE "(EMPSER)".
01  FNAME           PIC X(16) VALUE SPACES.
01  LFNAME          PIC 9(6) VALUE 16 COMP.
01  NFNAME          PIC X(7) VALUE "(FNAME)".
01  LNAME           PIC X(16) VALUE SPACES.
01  LLNAME          PIC 9(6) VALUE 16 COMP.
01  NLNAME          PIC X(7) VALUE "(LNAME)".
01  FNTYPE          PIC X    VALUE SPACES.
01  LFNTYPE         PIC 9(6) VALUE 1 COMP.
01  NFNTYPE         PIC X(3) VALUE "(F)".
01  SETMSG          PIC X(6) VALUE "SETMSG".
01  ERRMSG          PIC X(8) VALUE "MSG002  ".
01  GOODMSG         PIC X(8) VALUE "MSG001  ".
01  DISPSERV        PIC X(8) VALUE "DISPLAY ".
01  MENUPAN         PIC X(8) VALUE "MENUPAN ".
01  NAMEPAN         PIC X(8) VALUE "NAMEPAN ".
01  EMPLTBL         PIC X(8) VALUE "EMPLTBL ".
01  VDEFINE         PIC X(8) VALUE "VDEFINE ".
01  VRESET          PIC X(8) VALUE "VRESET  ".
01  CHAR            PIC X(8) VALUE "CHAR    ".
01  TBCREATE        PIC X(8) VALUE "TBCREATE".
01  TBGET           PIC X(8) VALUE "TBGET   ".
01  TBADD           PIC X(8) VALUE "TBADD   ".
01  TBPUT           PIC X(8) VALUE "TBPUT   ".
01  TBDELETE        PIC X(8) VALUE "TBDELETE".
01  TBOPEN          PIC X(8) VALUE "TBOPEN  ".
01  TBCLOSE         PIC X(8) VALUE "TBCLOSE ".
01  TABVARS         PIC X(13) VALUE "(FNAME LNAME)".
PROCEDURE DIVISION.
    CALL "ISPLINK" USING VDEFINE NFNAME FNAME CHAR LFNAME.
    CALL "ISPLINK" USING VDEFINE NLNAME LNAME CHAR LLNAME.
    CALL "ISPLINK" USING VDEFINE NEMPSER EMPSER CHAR LEMPSER.
    CALL "ISPLINK" USING VDEFINE NFNTYPE FNTYPE CHAR LFNTYPE.
    CALL "ISPLINK" USING TBOPEN EMPLTBL.
    IF RETURN-CODE NOT = 0 THEN
         CALL "ISPLINK" USING TBCREATE EMPLTBL NEMPSER TABVARS.
    CALL "ISPLINK" USING DISPSERV MENUPAN.
    PERFORM DISPMENU UNTIL RETURN-CODE = 8 OR FNTYPE > 4.
    CALL "ISPLINK" USING TBCLOSE EMPLTBL.
    CALL "ISPLINK" USING VRESET.
    STOP RUN.
```

(Continued on next page)

```
DISPMENU.
    CALL "ISPLINK" USING TBGET EMPLTBL.
    IF   FNTYPE = 1 AND RETURN-CODE NOT EQUAL 0 THEN
        MOVE SPACES TO FNAME
        MOVE SPACES TO LNAME
        PERFORM CONTRUN
    ELSE IF   FNTYPE > 1 AND RETURN-CODE EQUAL 0 THEN
        PERFORM CONTRUN
    ELSE
        CALL "ISPLINK" USING SETMSG ERRMSG.
    CALL "ISPLINK" USING DISPSERV MENUPAN.
CONTRUN.
    CALL "ISPLINK" USING SETMSG GOODMSG.
    IF   FNTYPE = 3 THEN
        CALL "ISPLINK" USING TBDELETE EMPLTBL
    ELSE
        CALL "ISPLINK" USING DISPSERV NAMEPAN
        IF   FNTYPE = 1 THEN
            CALL "ISPLINK" USING TBADD EMPLTBL
        ELSE IF   FNTYPE = 2 THEN
            CALL "ISPLINK" USING TBPUT EMPLTBL.
```

Appendix B. Complete FORTRAN Program Examples

Simple FORTRAN Program

The following is the simple FORTRAN program used in "Chapter 2: Developing Programs Using CMS" on page 19, as well as in "Chapter 10: Testing and Debugging Programs under VM/SP" on page 233.

```
      PROGRAM MYPROG
      CHARACTER*8 F,S
      WRITE (6,5)
      READ (5,2) F
      WRITE (6,10)
      READ (5,2) S
      WRITE (6,15) F,S
2     FORMAT (A8)
5     FORMAT (' ENTER YOUR FIRST NAME.')
10    FORMAT (' AND NOW YOUR LAST NAME.')
15    FORMAT (' WELCOME TO CMS, ',A8,1X,A8)
      STOP
      END
```

Complete FORTRAN Program

The following FORTRAN program lets the user add, change, delete, or display records in a file of peoples' names, by serial number. Records must be added before they can be changed, deleted or displayed.

FORT1 should be compiled and put in MODULE form, created by the GENMOD command:

```
GLOBAL TXTLIB VSF2FORT CMSLIB
GLOBAL LOADLIB VSF2LOAD
LOAD FORT1
GENMOD FORT1
```

EXEC for Complete FORTRAN Program

The following EXEC called DRIVE2, is an example of the type of procedure that you might use to drive the program given in "FORTRAN Program" on page 279.

```
&TRACE
STATE WORK DATA A
&IF &RETCODE GT 0 &GOTO -OK
&TYPE FILE 'WORK DATA A' EXISTS.  ERASE AND TRY AGAIN
&EXIT
-OK
FILEDEF NAMES DISK NAMES DATA
FILEDEF WORK DISK WORK DATA
FORT1
&IF &RETCODE NE 0 &GOTO -NG
ERASE NAMES DATA A
RENAME WORK DATA A NAMES DATA A
&EXIT
-NG
ERASE WORK DATA A
&EXIT
```

DRIVE2 invokes the program FORT1. It does the file management for FORT1 using CMS commands. The program creates a temporary work file, so DRIVE2 checks if the file already exists. If so, it issues an error message, and does not call the program. Otherwise it issues the FILEDEF commands and calls FORT1. Upon return, DRIVE2 tests the return code set by FORT1. If the return code indicates an incomplete work file, it is erased. If the return code indicates a completed work file, the old master file is erased and the work file renamed as the new master file.

The program FORT1 creates a master employee file. It reads the old master file and uses it as a base. It writes the modified file as a temporary work file. If there is an input error, this file will be incomplete, so it passes the status of it as a return code when it returns to the caller. A return code of 20 indicates an error condition, and therefore an incomplete file. You can manage these files manually. To do this, you must make sure that no file named WORK FILE A exists before you invoke the program. You must also issue the FILEDEF commands. Then, after the program has executed, you must inspect the return code. If it is 10 or 20, you must erase the work file. If it is 0, you must erase the master data file, and rename the work file as the master data file.

FORTRAN Program

```
      IMPLICIT INTEGER (A-Z)
      CHARACTER*6  EMPSER,SERNO
      CHARACTER*16 FNAME,LNAME,BNAME
      CHARACTER*21 MSGOK,MSGNG
      DATA  BNAME /'                '/
      DATA  MSGOK /'1OPERATION COMPLETED.'/
      DATA  MSGNG /'1INCORRECT SERIAL NO.'/
      FOUND = 0
      ENDSW = 0
100   FORMAT ('1ENTER FUNCTION NUMBER ')
200   FORMAT (' (1-ADD, 2-CHANGE, 3-ERASE, 4-DISPLAY, 5-END)')
300   FORMAT (' & REQUIRED SERIAL NO.')
400   FORMAT (I1,1X,A6)
500   FORMAT (A16,A16,A6)
600   FORMAT (A16)
700   FORMAT (' ENTER FIRST NAME:')
800   FORMAT (' ENTER LAST NAME:')
900   FORMAT (' ',A16,1X,A16)
1000  FORMAT (A21)
      WRITE (6,100)
      WRITE (6,200)
      WRITE (6,300)
      READ (6,400) FNTYPE,EMPSER
      IF (FNTYPE.GT.4) GO TO 70
      OPEN  (UNIT=11, FILE='NAMES')
      OPEN  (UNIT=12, FILE='DATA')
10    READ (11,500,ERR=75,IOSTAT=INT,END=15) SERNO,FNAME,LNAME
      IF (INT.NE.0) GO TO 75
      IF (EMPSER.EQ.SERNO) GO TO 20
      WRITE (12,500,ERR=75,IOSTAT=INT) SERNO,FNAME,LNAME
      GO TO 10
15    FOUND = 0
      ENDSW = 1
      GO TO 25
20    FOUND = 1
25    IF (FNTYPE.EQ.1.AND.FOUND.EQ.0) GO TO 30
      IF (FNTYPE.GT.1.AND.FOUND.EQ.1) GO TO 35
      WRITE (6,1000) MSGNG
      IF (FOUND.EQ.0) GO TO 65
      GO TO 55
30    FNAME = BNAME
      LNAME = BNAME
```

(Continued on next page)

```
      GO TO 45
35    IF (FNTYPE.EQ.3) GO TO 40
      GO TO 45
40    WRITE (6,1000) MSGOK
      GO TO 60
45    WRITE (6,900) FNAME,LNAME
      IF (FNTYPE.EQ.4) GO TO 50
      SERNO = EMPSER
      WRITE (6,700)
      READ (5,600) FNAME
      WRITE (6,800)
      READ (5,600) LNAME
50    WRITE (6,1000) MSGOK
55    WRITE (12,500,ERR=75,IOSTAT=INT) SERNO,FNAME,LNAME
      IF (ENDSW.EQ.1) GO TO 65
60    READ (11,500,ERR=75,IOSTAT=INT,END=65) SERNO,FNAME,LNAME
      IF (INT.EQ.0) GO TO 55
65    CLOSE (UNIT=11)
      CLOSE (UNIT=12)
      STOP
70    STOP 10
75    STOP 20
      END
```

Complete FORTRAN Program Using ISPF

The following FORTRAN program (FORT2) does the same things as the program in the previous section, but it uses ISPF.

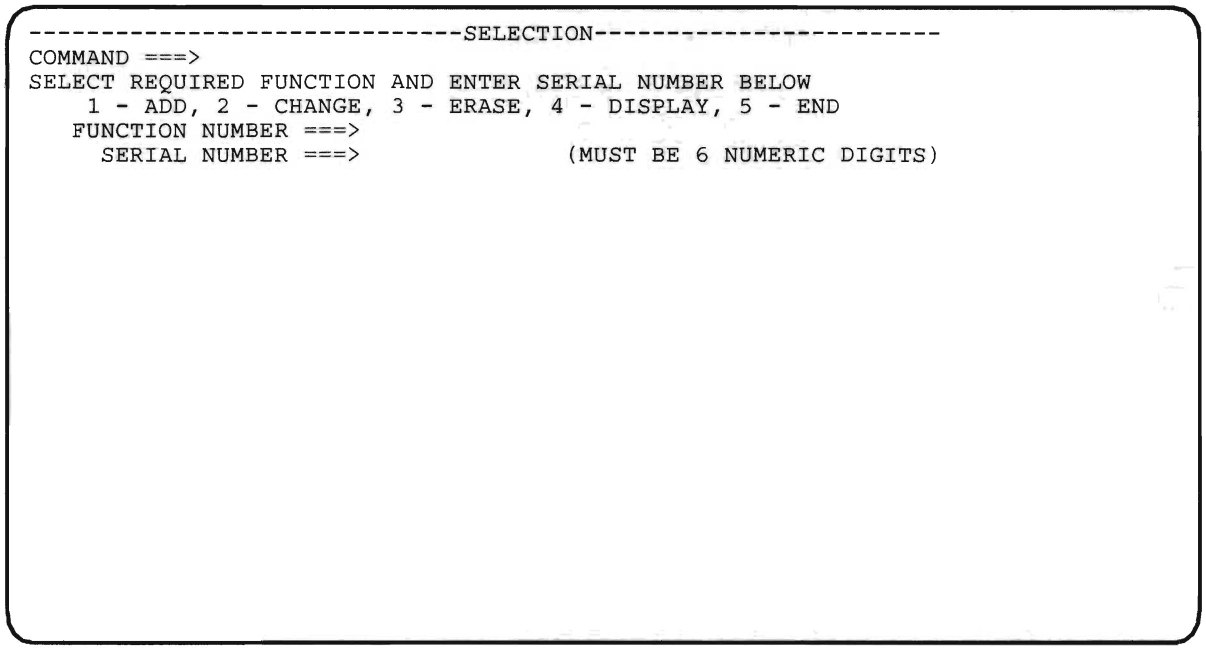
```
      IMPLICIT INTEGER (A-Z)
      CHARACTER*1  FNTYPE,FBLNK
      CHARACTER*6  EMPSER,EMPBLK
      CHARACTER*16 FNAME,LNAME,NAMEBL
      DATA  NAMEBL /'                '/
      DATA  EMPBLK /'      '/
      DATA  FBLNK /' '/
      LASTRC = ISPLNK ('VDEFINE','(FNAME)',FNAME,'CHAR',16)
      LASTRC = ISPLNK ('VDEFINE','(LNAME)',LNAME,'CHAR',16)
      LASTRC = ISPLNK ('VDEFINE','(EMPSER)',EMPSER,'CHAR',6)
      LASTRC = ISPLNK ('VDEFINE','(F)',FNTYPE,'CHAR',1)
      LASTRC = ISPLNK ('TBOPEN','EMPLTBL ')
      IF (LASTRC.EQ.0) GO TO 10
      LASTRC = ISPLNK ('TBCREATE','EMPLTBL ','(EMPSER)',
     * '(LNAME FNAME)')
10    FNTYPE = FBLNK
      EMPSER = EMPBLK
      LASTRC = ISPLNK ('DISPLAY','MENUPAN ')
      IF (LASTRC.EQ.8) GO TO 70
      IF (FNTYPE.GT.'4') GO TO 70
      LASTRC = ISPLNK ('TBGET','EMPLTBL ')
      IF (FNTYPE.EQ.'1'.AND.LASTRC.NE.0) GO TO 20
      IF (FNTYPE.GT.'1'.AND.LASTRC.EQ.0) GO TO 30
      LASTRC = ISPLNK ('SETMSG','MSG002  ')
      GO TO 10
20    FNAME = NAMEBL
      LNAME = NAMEBL
30    LASTRC = ISPLNK ('SETMSG','MSG001  ')
      IF (FNTYPE.EQ.'3') GO TO 40
      LASTRC = ISPLNK ('DISPLAY','NAMEPAN ')
      IF (FNTYPE.EQ.'1') GO TO 50
      IF (FNTYPE.EQ.'2') GO TO 60
      GO TO 10
40    LASTRC = ISPLNK ('TBDELETE','EMPLTBL ')
      GO TO 10
50    LASTRC = ISPLNK ('TBADD','EMPLTBL ')
      GO TO 10
60    LASTRC = ISPLNK ('TBPUT','EMPLTBL ')
      GO TO 10
70    CONTINUE
      LASTRC = ISPLNK ('TBCLOSE','EMPLTBL ')
      LASTRC = ISPLNK ('VRESET  ')
      STOP
      END
```

Appendix C. ISPF Panels

This is the ISPF specification of MENUPAN:

```
)BODY
%------------------------------SELECTION------------------------
%COMMAND ===>_ZCMD                                             +
+
+
%SELECT REQUIRED FUNCTION AND ENTER SERIAL NUMBER BELOW
+
+
+   1 - ADD, 2 - CHANGE, 3 - ERASE, 4 - DISPLAY, 5 - END
+
+
+  FUNCTION NUMBER%===>_FNTYPE+
+
+
+    SERIAL NUMBER%===>_EMPSER+      (MUST BE 6 NUMERIC DIGITS)
+
+
+
)INIT
  .CURSOR = F
)PROC
  VER (&F, NONBLANK)
  VER (&F, PICT,N)
  IF (&F ¬=5)
    VER (&EMPSER, NONBLANK)
    VER (&EMPSER, PICT,NNNNNN)
)END
```

This is the way it's displayed on the screen:

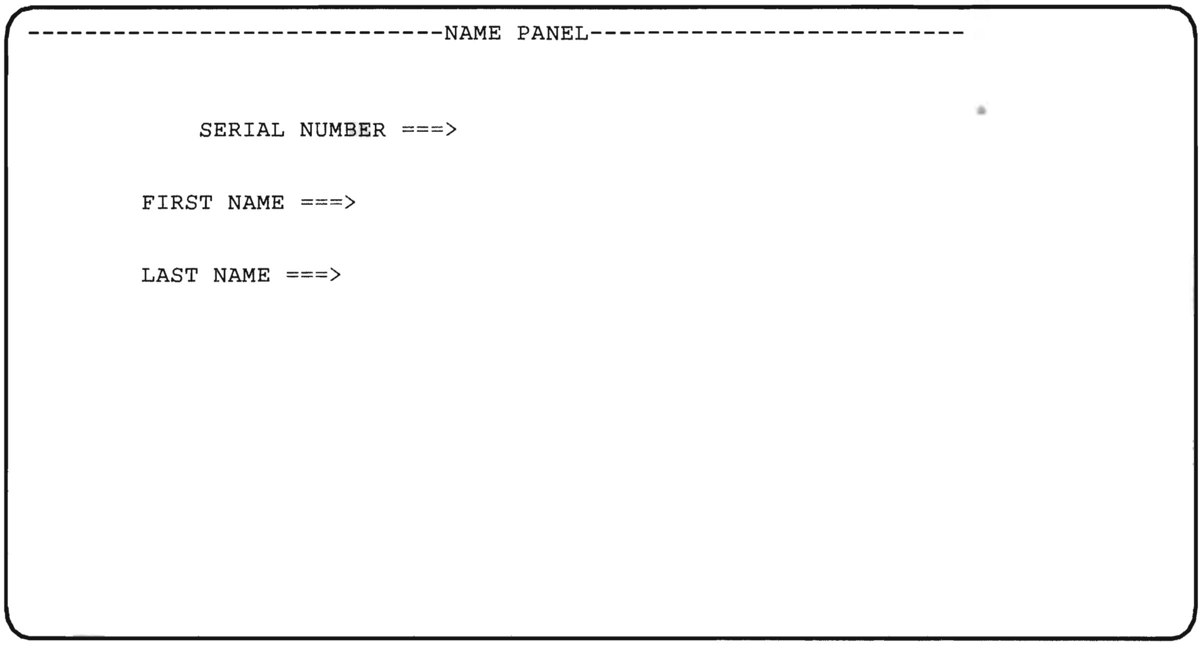


```
------------------------------SELECTION-------------------------
COMMAND ===>
SELECT REQUIRED FUNCTION AND ENTER SERIAL NUMBER BELOW
    1 - ADD, 2 - CHANGE, 3 - ERASE, 4 - DISPLAY, 5 - END
   FUNCTION NUMBER ===>
     SERIAL NUMBER ===>             (MUST BE 6 NUMERIC DIGITS)
```

This is the ISPF specification of NAMEPAN:

```
)BODY
%----------------------------NAME PANEL--------------------------
+
+
+
+           SERIAL NUMBER%===>_EMPSER+
+
+
+       FIRST NAME%===>_FNAME            +
+
+
+       LAST NAME%===>_LNAME             +
+
+
+
)PROC
  .CURSOR = FNAME
  VER (&FNAME,ALPHA)
  VER (&LNAME,ALPHA)
)END
```

This is the way it's displayed on the screen:



```
----------------------------NAME PANEL--------------------------


            SERIAL NUMBER ===>

        FIRST NAME ===>

        LAST NAME ===>
```

These are the ISPF specifications of the two messages issued by the COBOL and FORTRAN programs:

```
MSG001   'OPERATION COMPLETED'                      .ALARM=NO
'THE OPERATION SPECIFIED HAS BEEN COMPLETED.'
MSG002   'INVALID OPERATION'                        .ALARM=YES
'ENTER A NUMBER FROM 1 TO 5 IN THE SPACE PROVIDED.'
```

These are the filedefs to be issued prior to running the example programs COBOL2 and FORT2.

```
FILEDEF ISPPROF DISK ISPPROF MACLIB A (PERM
FILEDEF ISPPLIB DISK USERPAN MACLIB * (PERM CONCAT
FILEDEF ISPPLIB DISK ISRPLIB MACLIB * (PERM CONCAT
FILEDEF ISPPLIB DISK ISPPLIB MACLIB * (PERM CONCAT
FILEDEF ISPMLIB DISK EXAMMSG MACLIB * (PERM CONCAT
FILEDEF ISPMLIB DISK ISRMLIB MACLIB * (PERM CONCAT
FILEDEF ISPMLIB DISK ISPMLIB MACLIB * (PERM CONCAT
FILEDEF ISPSLIB DISK ISRSLIB MACLIB * (PERM CONCAT
FILEDEF ISPTABL DISK MYTABLE MACLIB A (PERM
FILEDEF ISPTLIB DISK MYTABLE MACLIB * (PERM CONCAT
FILEDEF ISPTLIB DISK ISRTLIB MACLIB * (PERM CONCAT
FILEDEF ISPTLIB DISK ISPTLIB MACLIB * (PERM CONCAT
FILEDEF ISPXLIB DISK FORTVS2  TXTLIB * (PERM CONCAT
FILEDEF ISPXLIB DISK COBOLVS  TXTLIB * (PERM CONCAT
FILEDEF ISPXLIB DISK COBLIBVS TXTLIB * (PERM CONCAT
FILEDEF ISPXLIB DISK MYLIB    TXTLIB * (PERM CONCAT
```

Summary of Changes

Summary of Changes
for SC24-5247-2
for VM/SP Release 5

Miscellaneous

Minor changes have been made to this publication to reflect changes caused by the following VM/SP Release 5 enhancements:

- Addition of the Session Manager
- 3270 Usability Enhancement
- Central Message Facility for NLS
- TXTLIB Enhancement

This major revision also incorporates other minor technical and editorial changes.

Summary of Changes
for SC24-5247-1
for VM/SP Release 4

VS FORTRAN Version 2 Compiler Support

Changes have been made to this publication to reflect support for the VS FORTRAN Version 2 Compiler.

HPO Vector Facility Support

Changes have been made to this publication to reflect support for the HPO Vector Facility. The VM/SP HPO Vector Facility is an instruction processor that can manipulate values (usually floating-point values) at a high speed. The HPO Vector Facility is supported by the VS FORTRAN Version 2 program.

Miscellaneous

Minor technical and editorial changes have also been made to this publication.

Bibliography

Consult the following books for more information on specific subjects:

- General

  IBM System/370 Principles of Operation, GA22-7000

- VM/SP

  VM/SP General Information, GC20-1838

  VM/SP CP Command Reference, SC19-6211

  VM/SP Introduction, GC19-6200

  VM/SP Terminal Reference, GC19-6206

- CMS

  VM/SP CMS Command Reference, SC19-6209

  VM/SP Macros and Functions Reference, SC24-5284

  VM/SP CMS User's Guide, SC19-6210

- COBOL

  IBM OS COBOL Interactive Debug Terminal User's Guide and Reference, SC28-6465

  CMS User's Guide for COBOL, SC28-6469

- FORTRAN

  VS FORTRAN Version 2 Interactive Debug Guide and Reference, SC26-4223

  VS FORTRAN Version 2 Application Programming Guide, SC26-4222

- System Product Editor

  VM/SP System Product Editor Command and Macro Reference, SC24-5221

- ISPF

  ISPF Dialog Management Services and Examples, SC34-4010

  ISPF Dialog Management Guide, SC34-4009

  ISPF/PDF for VM/SP Guide, SC34-4011

- DMS/CMS

Display Management System for CMS: Guide and Reference, GC24-5198

- SQL/DS

  SQL/DS Application Programming, SH24-5068

  SQL/DS Concepts and Facilities, SH24-5065

  SQL/DS Planning and Administration — VM/SP, SH24-5043

  SQL/DS Terminal User's Guide — VM/SP, SH24-5045

- System Product Interpreter

  VM/SP System Product Interpreter User's Guide, SC24-5238

  VM/SP System Product Interpreter Reference, SC24-5239

- EXEC2

  VM/SP EXEC2 Reference, SC24-5219

- VM/SP HPO

  VM/SP HPO General Information, GC19-6221

  VM/SP HPO CP Command Reference for General Users, SC19-6227

  VM/SP HPO Operator's Guide, SC19-6225

Note: This manual assumes that VS FORTRAN Version 2 or COBOL VS is used, except where explicitly stated otherwise.

The VM/SP Library (Part 1 of 3)

Evaluation

- General Information — GC20-1838
- Introduction — GC19-6200

Index

- Library Guide, Glossary, and Master Index — GC19-6207

Planning

- Planning Guide and Reference — SC19-6201
- Running Guest Operating Systems — GC19-6212
- Release 5 Guide — SC24-5290
- Distributed Data Processing Guide — SC24-5241

Installation

- Installation Guide — SC24-5237

Applications

- Application Development Guide — SC24-5247
- Programmer's Guide to the SRPI for VM/SP — SC24-5291

Operation

- Operator's Guide — SC19-6202

Reference Summaries

To order all of the Reference Summaries, use order number SBOF-3242

- Commands (General User) — SX20-4401
- Commands (Other than General User) — SX20-4402
- SP Editor Command Reference Summary — SX24-5122
- EXEC 2 Reference Summary — SX24-5124
- Sys.Prod Interpreter Reference Summary — SX24-5126
- CMS Primer Summary of Commands — SX24-5151
- CMS Primer Line-Oriented Summary of Commands — SX24-5159
- Problem Reporting Summary (Poster) — SX24-5171
- Summary of End Use Tasks and Commands (Poster) — SX24-5173

The VM/SP Library (Part 2 of 3)

End Use

Terminal Reference	GC19-6206
CMS Primer	SC24-5236
CMS Primer for Line-Oriented Terminals	SC24-5242
CMS User's Guide	SC19-6210
CMS Command Reference	SC19-6209
CMS Macros and Functions Reference	SC24-5284
System Product Editor User's Guide	SC24-5220
System Product Editor Command and Macro Reference	SC24-5221
System Product Interpreter User's Guide	SC24-5238
System Product Interpreter Reference	SC24-5239
EXEC 2 Reference	SC24-5219
CP Command Reference	SC19-6211
Quick Reference	SX20-4400

Diagnosis

System Messages and Codes	SC19-6204
System Messages Cross-Reference	SC24-5264
Service Routines Program Logic	LY20-0890
Problem Reporting Guide	SC24-5282
VM Diagnosis Guide	LY24-5241
GCS Diagnosis Reference	LY24-5239
Problem Determination Vol. 1 (CP)	LY20-0892
Data Areas and Control Blocks Vol. 1 (CP)	LY24-5220
Problem Determination Vol. 2 (CMS)	LY20-0893
Data Areas and Control Blocks Vol. 2 (CMS)	LY24-5221
OLTSEP and Error Recording Guide	SC19-6205
VM Problem Determination Reference Information	LX23-0347
VM CP Internal Trace Table (Poster)	LX24-5202

The VM/SP Library (Part 3 of 3)

Administration

- VM System Facilities for Programming SC24-5288
- CP for System Programming SC24-5285
- CMS for System Programming SC24-5286
- TSAF Reference SC24-5287
- GCS Command and Macro Reference SC24-5250

Auxiliary Communication Support

- VTAM Installation and Resource Definition SC23-0111
- VTAM Customization SC23-0112
- VTAM Operation SC23-0113
- VTAM Messages and Codes SC23-0114
- VTAM Reference Summary SC23-0135
- VTAM Programming SC23-0115
- VTAM Diagnosis Guide SC23-0116
- VTAM Diagnosis Reference LY30-5582
- VTAM Data Areas (VM) LY30-5583
- RSCS Networking Version 2 General Information GH24-5055
- RSCS Networking Version 2 Planning and Installation SH24-5057
- RSCS Networking Version 2 Operation and Use SH24-5058
- RSCS Networking Version 2 Diagnosis Reference LY24-5228
- RSCS Networking Version 2 Ref. Summary SX24-5135
- VM/Pass-Through Facility General Information GC24-5206
- VM/Pass-Through Facility Guide and Reference SC24-5208
- VM/Pass-Through Facility Logic LY24-5208

Index

International Business
Machines Corporation
P.O. Box 6
Endicott, New York 13760

File No. S370/4300-50
Printed in U.S.A.

SC24-5247-2

VM/SP
Application Development Guide
Order No. SC24-5247-2

READER'S COMMENT FORM

Is there anything you especially like or dislike about this book? Feel free to comment on specific errors or omissions, accuracy, organization, or completeness of this book.

If you use this form to comment on the online HELP facility, please copy the top line of the HELP screen.

__________ __________ ______ Help Information line ___ of ___

Would you like a reply? ___YES ___NO

Please print your name, company name, and address:

____

____

____

____

IBM Branch Office serving you: ________________________

Thank you for your cooperation. You can either mail this form directly to us or give this form to an IBM representative who will forward it to us.

SC24-5247-2

Reader's Comment Form

CUT
OR
FOLD
ALONG
LINE

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE:

INTERNATIONAL BUSINESS MACHINES CORPORATION
DEPARTMENT G60
PO BOX 6
ENDICOTT NY 13760-9987

Fold and tape | Please Do Not Staple | Fold and tape

®

VM/SP
Application Development Guide
Order No. SC24-5247-2

READER'S COMMENT FORM

Is there anything you especially like or dislike about this book? Feel free to comment on specific errors or omissions, accuracy, organization, or completeness of this book.

If you use this form to comment on the online HELP facility, please copy the top line of the HELP screen.

______________ ______________ _______ **Help Information line ____ of ____**

Note: Do not use this form to report system problems or to request copies of publications. Instead, contact your IBM representative or the IBM branch office serving you.

Would you like a reply? ___YES ___NO

Please print your name, company name, and address:

____

____

____

____

IBM Branch Office serving you: ____________________________________

Thank you for your cooperation. You can either mail this form directly to us or give this form to an IBM representative who will forward it to us.

SC24-5247-2

Reader's Comment Form

CUT
OR
FOLD
ALONG
LINE

Fold and tape — Please Do Not Staple — Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM

INTERNATIONAL BUSINESS MACHINES CORPORATION
DEPARTMENT G60
PO BOX 6
ENDICOTT NY 13760-9987

Fold and tape — Please Do Not Staple — Fold and tape

IBM ®

International Business
Machines Corporation
P.O. Box 6
Endicott, New York 13760

File No. S370/4300-50
Printed in U.S.A.

SC24-5247-2

SC24-5247-02