DataGeneral

Customer Documentation

Learning to Use Your AOS/VS System

Learning to Use Your AOS/VS System

069-000031-02

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 069-000031

Revision 02, February 1986

NOTICE

Learning to Use Your
AOS/VS System
069-000031

Revision History:	Effective with:
Original Release - October 1981	
First Revision - April 1984	
Second Revision - February 1986	AOS/VS Rev. 7.00

A vertical bar in the margin of the Contents indicates substantive change from the previous revision.

Preface

The Manual's Objectives

Learning to Use Your AOS/VS System introduces the AOS/VS Operating System and its interface, the Command Line Interpreter (CLI). You'll learn, in an interactive session, how to perform everyday operations under AOS/VS. And the manual includes a command dictionary so that you can check CLI commands and syntax as you get started.

After providing some hands-on experience with AOS/VS, the manual introduces you to different products — languages and utilities — that run under AOS/VS. Use these chapters as you would a cookbook — select only the sections that interest you.

Learning to Use AOS/VS shows you how to create and edit text files with the SED or SPEED editors. It sketches the program development steps required by different AOS/VS languages. Many languages — BASIC, Business BASIC, C, COBOL, Interactive COBOL, FORTRAN 77, and Pascal — have their own sample programming session. The manual also describes major features of Data General's Assembler language. In the course of developing programs in each language, you'll learn how to use different debuggers that run under AOS/VS. Finally, the manual shows you how to use the Sort/Merge Utility and explains AOS/VS record formats.

Once you've completed the hands-on sessions, we suggest you turn to the Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS), the definitive work on AOS/VS. And for further learning, see the documentation guide in Chapter 16. It describes the documentation available for AOS/VS products, so you can chart your course from here.

Who Are You?

Most of the readers of this manual are programmers or other professionals who know something about computers and operating systems, but who are unfamiliar with AOS/VS and other Data General software. If you're primarily interested in CEO® (Comprehensive Electronic Office), our office automation product, then *Getting Started with the CEO® System* is the manual for you. Or, if computers are new to you, see your system manager for a basic introduction, and then turn to the hands-on session in Chapter 2.

The examples assume that you're working in a timesharing environment to write source files or documents, run applications, and — in many cases — develop programs. The language chapters assume that you already know how to program in a particular language, and simply require an overview of Data General's implementation of the language.

Using the Manual

Chapter	What It's About	Who Should Read It?
1	AOS/VS as an operating system, the features of AOS/VS, its directory structure, and program development in general.	Anyone who wants to see the big picture.
2	An interactive session with AOS/VS, showing everyday procedures — from logging on to printing files.	Everyone.
3	A reference chapter, highlighting CLI syntax and control sequences and describing common CLI commands.	Everyone, on occasion.
4	The SED text editor, including a hands-on session in which you create and edit a text file.	Anyone interested in text editing.
5	The SPEED text editor, a character-oriented editor that interests many programmers because it can execute commands conditionally and edit invisible characters.	Anyone interested in a character- oriented editor.
6	Developing, running, and compiling an AOS/VS BASIC program.	An AOS/VS BASIC programmer.
7	Developing and running a Business BASIC program.	A Business BASIC programmer.
8	Developing, running, and debugging a C program.	A C programmer.
9	Developing, running, and debugging a COBOL program.	A COBOL programmer.
10	Developing, running, and debugging an Interactive COBOL program.	An Interactive COBOL programmer.
11	Developing, running, and debugging a FORTRAN 77 program.	A FORTRAN 77 programmer.
12	Developing, running, debugging, and optimizing a Pascal program.	A Pascal programmer.
13 and 14	An overview of assembly language programming, showing how to develop and run a program in Data General Assembler.	An Assembler programmer.
15	An explanation of AOS/VS record formats and a sample session showing how to use the Sort/Merge Utility.	Everyone.
16	Selecting and ordering documentation on AOS/VS and its related products.	Everyone.
Glossary	Terminology used in this manual (or common to the computer industry).	Everyone on occasion.

Reader Please Note

The manual uses the following symbols:

⤶	Press the NEW LINE key on your keyboard.
)	is the CLI prompt.
CTRL-letter	Hold down the CTRL key, and press a second key. This is called a control sequence.

Most examples in this manual are interactive: that is, you type commands on the keyboard, and the system displays or prints responses or prompts on the screen (or hardcopy terminal). **This typeface identifies what you type**, ***and this typeface identifies the system messages and prompts***.

The manual uses both uppercase and lowercase characters, but you can use whichever you choose. Certain languages, such as COBOL and Business BASIC, require uppercase characters, so the examples in those chapters are uppercase.

All numbers are decimal unless we indicate otherwise: e.g., 35_8.

We use these conventions for command formats in this manual:

COMMAND required *[optional]* ...

Where	**Means**
COMMAND	You must enter the command (or its accepted abbreviation) as shown.
required	You must enter some argument (such as a filename). Sometimes, we use: $\begin{Bmatrix} \text{required}_1 \\ \text{required}_2 \end{Bmatrix}$ which means you must enter *one* of the arguments. Don't enter the braces; they only set off the choice.
[optional]	You have the option of entering this argument. Don't enter the brackets; they only set off what's optional.
...	You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.

Contacting Data General

- If you have comments on this manual, please use the prepaid Remarks Form that appears after the Index. We want to know what you like and dislike about this manual.
- If you need additional manuals, please use the enclosed TIPS order form (USA only) or contact your Data General sales representative.

End of Preface

Contents

Chapter 1 - An Introduction to AOS/VS

Chapter 2 - A Session with AOS/VS

Chapter 3 - Common CLI Commands

Chapter 4 - Writing with the SED Text Editor

Chapter 5 - Writing with the SPEED Text Editor

Chapter 6 - AOS/VS BASIC Programming

Chapter 7 - Business BASIC Programming

Chapter 8 - C Programming

Chapter 9 - COBOL Programming

Chapter 10 - Interactive COBOL Programming

Chapter 11 - FORTRAN 77 Programming

Chapter 12 - Pascal Programming

Chapter 13 - Assembly Language Programming

Chapter 14 - Writing AOS/VS Assembly Language Programs

Chapter 15 - The Sort/Merge Utility

Chapter 16 - Documentation Guide

Glossary

Index

Tables

Table

Illustrations

Figure

Chapter 1
An Introduction to AOS/VS

This chapter provides an overview of AOS/VS and operating systems. It describes how AOS/VS works, and introduces terms that you'll see later on in the book. For those new to operating systems, the chapter will explain the environment in which you'll work. If you're new to AOS/VS, but not to operating systems, the chapter will tell you about the unique features of AOS/VS and provide some translation from the operating system with which you're familiar. The major sections of the chapter include

- What is an Operating System?
- Special Features of AOS/VS
- Building Programs Under AOS/VS
- Transporting Programs to AOS/VS

You don't need to understand all the concepts in the chapter to get going. Chapters 2, 4, and 5 provide hands-on sessions that will get you started on AOS/VS. You can always return to this chapter later.

What is an Operating System?

An operating system is a large program that serves as a bridge between users of a computer and the computer hardware itself. At its most basic level, an operating system translates a user's commands into the language of the machine, which the hardware can process, and then reports the results — translated again — back to the user.

How does this communication occur? Physically, a computer system consists of a processor, a system console, and a disk. Most often, the system will also have several more disks, tape units, line printers, and user terminals (see Figure 1-1). Sometimes this hardware will be available to the people using a computer; other people may only have access to a terminal and its keyboard.

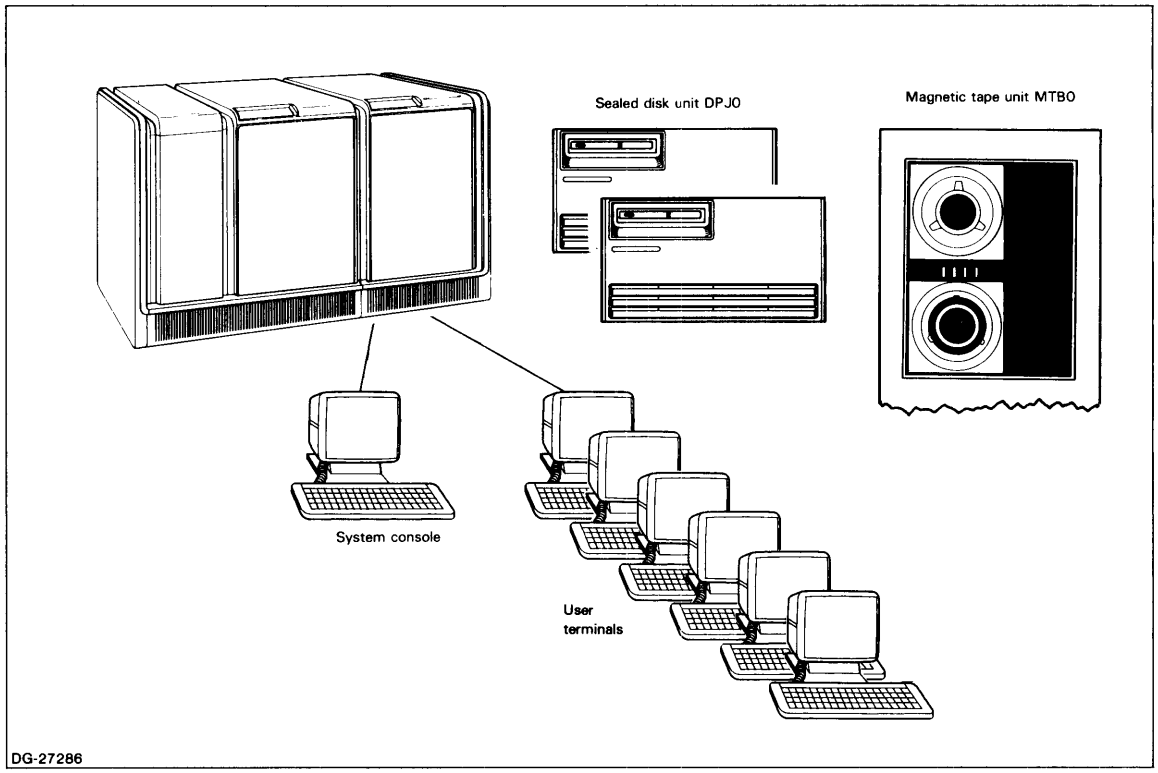


Figure 1-1. An AOS/VS Computer System

The hardware shown in Figure 1-1 is all electronically connected, with the operating system transferring information from one device to another. Any person authorized to use the system can enter commands on a terminal keyboard, such as the one shown in Figure 1-2; and the operating system will translate the commands into directives for the processor (or any other system device). The processor completes each task and relays the results to the operating system when it's done. The operating system, in turn, displays a message or provides some indication to the user that the task is complete.

A timesharing computer system is not available to everyone. The operating system recognizes authorized users, who have accounts on the system, and extends computer resources to them according to their account profile. Thus, the operating system manages the variety of privileges available to users. Those without permission cannot start a session or engage the computer in any way.

PH-0806

Figure 1-2. DASHER® D460 Terminal

In addition to relaying information from one hardware device to another, an operating system provides many basic computer services that system users and computer programs routinely require. The operating system manages data and keeps it secure; it allocates resources such as the tape unit or line printer; it can also deliver mail and provide tools for program development. In this way users and programs can draw on a "toolbox" of services, which relieves them of having to include minute instructions for every task their programs might require.

Another role of an operating system is supervisory: it oversees and logs the activity of all system hardware, and the processor in particular. The operating system budgets the processor's time, so that each job receives its fair share of processing time. An operating system reviews incoming requests for computer services and then allocates resources — such as the processor, physical memory, or a tape unit — as they become available. An operating system maintains security — it isolates users from one another and makes sure that each user's work is safe from intrusion and causes no harm.

The number of people and programs that an operating system can support varies. Some operating systems support many users, whereas others are intended for only one or two users. Whatever the case, a well-designed operating system should serve its users in such a way that they feel they're the only ones using the computer.

Processes

The operating system packages all executing programs into jobs called *processes*. Each process is like a complete computer system. It consists of a group of instructions, called a program file, plus a work space in computer memory, and permission to use system resources such as the processor, a keyboard, and a printer. When the operating system creates a process, it attaches an identifying number to the process and, thereafter, refers to the program request with the process ID (or PID) number.

Processes are temporary entities — the operating system creates them when a user executes a program, and terminates them when the program ends.

There are many different kinds of processes: some are created by the system and related to the daily operation of the computer, and others are created for system users.

Special Features of AOS/VS

AOS/VS is a Data General operating system designed to run on the ECLIPSE® MV/Family computers — our 32-bit computers. AOS/VS can support hundreds of users and run hundreds of processes.

AOS/VS is a powerful system. Not only can it run many processes concurrently, but it can also handle programs of almost any size because it uses a composite of main memory and disk to house the program. AOS/VS gives each process at least 512 megabytes of address space, which can be expanded to 2048 megabytes. Processes can use any hardware device configured into the system.

AOS/VS runs programs and parts of programs (tasks) concurrently. This means that each process can perform several different tasks at the same time, and that each task can respond individually to its own environment. Multitasking can make a program more efficient by permitting it to do useful processing while waiting for a hardware device (like a tape unit, line printer, or disk) to complete an operation.

On the majority of sites, AOS/VS operates in two modes simultaneously: interactive and batch. When people use AOS/VS interactively, they work at terminals, entering instructions through their keyboard, and receiving operating system prompts or program results on the screen. When people use AOS/VS in batch, they don't have to be at their terminals. They submit the instructions ahead of time, with any other necessary data, and the operating system takes care of the job when it finds the time. This leaves users free to perform other tasks or run other programs. AOS/VS creates a process for each batch job and for each interactive process running from a user's terminal. Users can run several interactive and batch jobs at the same time, maximizing their use of the system.

Another characteristic of AOS/VS is its security. System users have their own work areas, and can define their files as public or private. Just as AOS/VS protection mechanisms make each user's work safe from intrusion, they also prevent a user's programming errors from harming the system or other processes. Moreover, a system manager can selectively assign privileges and system resources, such as disk space, to users according to their computing requirements.

So AOS/VS provides a bank of services to programs and its user community. Beyond this, AOS/VS supports a great number of application programs and utilities. It supports many high-level languages, such as Ada*, BASIC, Business BASIC, C, COBOL, Interactive COBOL, FORTRAN 5, FORTRAN 77, Pascal, and PL/I. It can also run 16- or 32-bit programs simultaneously. A number of text editors run under AOS/VS: SED, SPEED, and the CEO Word Processor. And the system supports a large number of utilities: Sort/Merge; several debuggers, such as SWAT, the AOS/VS Debugger, the Interactive COBOL Debugger; and file comparison utilities such as FILCOM, SCOM, and DISPLAY. In addition, there are many system management utilities (documented in *How to Generate and Run AOS/VS*), as well as database, networking, and office automation programs (documented in Chapter 16).

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

It's possible to convert programs from other Data General operating systems to run under AOS/VS because it is compatibile with the Advanced Operating System (AOS), the Advanced Operating System/Real Time 32 (AOS/RT32), the Micro Processor/Advanced Operating System (MP/AOS), and the Real-Time Disk Operating System (RDOS). Data General's AOS system, which runs on 16-bit ECLIPSE computers, was the basis for AOS/VS. Nearly all system programs work exactly the same way in AOS and AOS/VS, but some languages have different requirements. Programs written for AOS always need recompiling and relinking to run on AOS/VS. But it's possible to develop programs in many languages with AOS/VS that will compile and run — without change — on AOS systems.

The Command Line Interpreter (CLI)

A user communicates with the AOS/VS operating system through an interactive program called the Command Line Interpreter (CLI). AOS/VS sets up a CLI process as soon as a user logs on the computer, so that he or she can use the program's command language to obtain all kinds of services. Since the CLI command vocabulary consists of more than 100 commands, users can run programs and utilities, work with files, and submit jobs to such hardware devices as line printers and tape units.

Files

The AOS/VS operating system is basically a large cluster of program files. (A file is any independent set of instructions — or any collection of information treated as a unit.) Each utility provided by Data General, including the CLI, is a program file. Certain program files can read other files, act on what they read, and produce new, updated output files.

Even though the computer system that AOS/VS drives consists of hardware such as the processor, printers, tape units, or terminals — devices that are physically real — AOS/VS users don't address the devices directly. Instead, AOS/VS users always address the filename of the device. So, when AOS/VS receives a request for a printer or a tape unit, it places the request into a file, which is then passed to the device.

From a user's point of view, files are collections of information such as accounting figures, lists of phone numbers, or a text source file that forms the basis for a program. Files commonly reside on disk or tape, in the same way that folders reside in a file drawer, and are brought into computer memory when needed. A disk file can be very large or very small; the largest file can store 4 billion bytes; the smallest, 0 bytes. All files have unique names, and there are many different types of files, depending on their contents.

Filenames

Computer users always refer to files by name, which are assigned at the time they are created. AOS/VS filenames must contain between 1 and 31 of the following characters: upper or lowercase letters, numbers, the period, the dollar sign, the question mark, and the underscore. For example, TRANS_$INCOME_RECEIVED?MAY.10 is a valid, although cumbersome, filename.

Filenames are not case sensitive. The system translates lowercase characters into uppercase internally, so filename "test" is also "TEST" — but this is invisible to users at their terminals.

A suffix ("extension") attached to a filename helps identify the contents of the file. For example, FORTRAN 77 source filenames usually end in .F77. Using the last three characters in this way is not required, but it makes it easier to keep track of files.

Directories

There are two main types of files: directory files and data files. Directory files are like library card files or the drawers of a file cabinet because both help organize files. Each directory file contains the names of the files inside. (Each data file contains a cluster of information too, but not about files. A data file contains information like source code or text.)

System users can create directories at will to help organize their files. For example, a directory named FORTRAN could contain all FORTRAN files and a directory called PERSONAL could contain all personal files.

AOS/VS has a certain number of its own directories (see Figure 1-3). Notice that the directories are organized into a hierarchical structure and the highest, most important directory is the root (:). Subordinate to the : are the system directories for utilities, hardware devices, networking, and so on. The utilities directory, called UTIL, contains the system utilities, while the networking directory, NET, contains the names of computers in your network. One directory — UDD — is for system users; your username directory and its subdirectories are created within this directory.

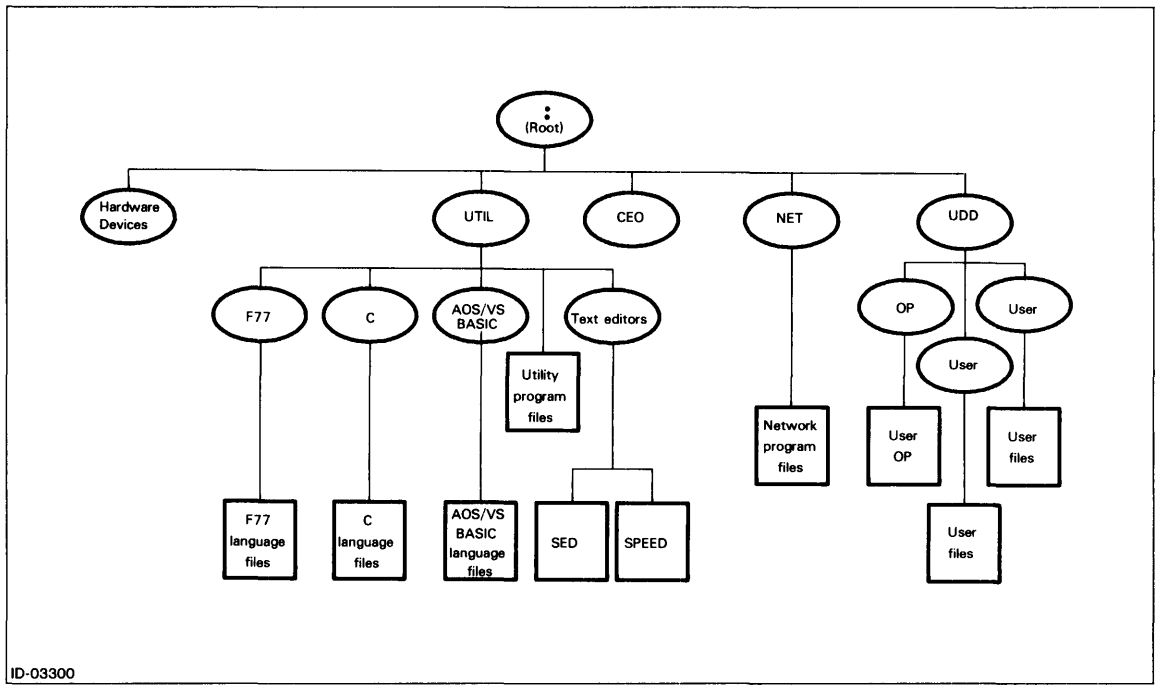


Figure 1-3. A Sample AOS/VS File Structure

Building Programs Under AOS/VS

This manual provides an overview of program development under AOS/VS. Many of the languages supported by AOS/VS — BASIC, Business BASIC, C, COBOL, Interactive COBOL, FORTRAN 77, and Pascal — have their own sample programming session. There are two chapters on Data General assembly language programming. Then, in Chapter 16, we describe the manuals available in each of these langauges, so you have a good idea of where to go next for information.

Generally, program development requires these five steps:

1. Creating a source program file using a text editor.
2. Compiling or assembling the source file(s) to produce an object module. This is done by the *compiler* or *assembler*, respectively, which are programs that translate source programs into binary machine code that the computer can understand.
3. Linking the object module(s) into an executable program file.
4. Running the program file.
5. Debugging the program file, if necessary. By executing the program from the debugger, it's possible to isolate and correct many program errors.

Transporting Programs to AOS/VS

You can transfer programs written on other operating systems to AOS/VS. After bringing the source files over, you sometimes need to recompile, but usually only need to relink the source programs before they'll run under AOS/VS. Cross-development documentation, available in the respective language manual, explains the transfer process in detail. We briefly describe the steps below.

From Other Data General Systems

In order to transport program files from another Data General operating system to AOS/VS, copy the program source files to tape on Data General tape units at the source site. Use the CLI DUMP comand to copy, not the COPY command.

For source programs dumped from an AOS/VS, AOS or MP/AOS system, you can simply mount the tape on a local tape unit, and use the CLI LOAD command to copy files onto disk. Mounting a tape is described in Chapter 2; the LOAD command is described in Chapter 3.

For source programs dumped from RDOS, DOS, or ICOS systems, mount the tape on a unit and use the RDOS utility to convert the sources. For example, to load and convert the file(s) in the first tape file of a tape mounted on unit 0, you would enter the command,

```
) xeq RDOS load/v @mtb0:0 +/c  ↵
```

The Command Line Interpreter (CLI) User's Manual describes the RDOS utility further.

From Other Manufacturers

Data General text editors and compilers often require ASCII files with data-sensitive records. The data-sensitive delimiters are ASCII

- NEW LINE (12_8 or $0A_{16}$)
- Form feed (14_8 or $0C_{16}$)
- Null (0).

So — if possible — the source site files should be in ASCII, with NEW LINE as a record delimiter. If the files are ASCII with NEW LINE as a delimiter, you can load the files with the CLI COPY command, described in Chapter 3, and edit them with a text editor.

But if the files are in EBCDIC or if records do not have data-sensitive delimiters, you need to convert the files. The easiest way to do this is with the Sort/Merge utility. Chapter 15 explains Sort/Merge, and provides an example of program conversion from EBCDIC to ASCII, with the insertion of NEW LINE delimiters.

What Next?

To learn how to log on, create files and directories, and use the line printer or the tape unit, turn to Chapter 2. In its hands-on session you'll gain experience with everyday CLI commands and procedures. If you are simply interested in using a text editor, then go directly to the SED or SPEED text editing session in Chapters 4 or 5, respectively. Those of you using only the Comprehensive Electronic Office (CEO), should turn to the *Getting Started with the CEO® System* manual. And some of you, those who are primarily interested in networking under AOS/VS, should refer to *Using the XODIAC™ Network Management System (AOS and AOS/VS).*

End of Chapter

Chapter 2
A Session with AOS/VS

The best way to learn AOS/VS is to use it, so this chapter leads you through a sample session. It consists of eleven sections, each demonstrating one aspect of the operating system. These sections are

- Taking the First Steps
- Getting Your Bearings
- Learning About Files
- Working with Directories
- Special Keys Under AOS/VS
- More About Files and Directories
- Printing Files
- Protecting Files
- Obtaining Help from the CLI
- Batch Processing
- Running Programs from the CLI

Each of the eleven sections consists of an overview, a series of exercises, and a summary. You can choose to cover the entire session in one sitting, or complete a few sections at a time. Whichever you do, don't hurry. A log file can record and summarize the complete session — we show how to open a log file, early in the session, and provide a sample log file in Figure 2-3. The Glossary explains special terminology.

AOS/VS is a secure system, with good error handling and explicit error messages. The Security Check List at the end of the chapter reminds you of practices that help maintain AOS/VS security. Any mistakes you make in the session will not harm the system or other users, so you can proceed with confidence.

Taking the First Steps

In order to start a session with AOS/VS, you need a terminal and an account on the system. You can obtain both of these from the person who's responsible for your system, often called a system manager. Once you turn on your terminal, you can begin a session with AOS/VS by identifying yourself as an authorized user.

This section shows you how to start and stop a computer session, and how to maintain a secure account on the system.

If you've never used a computer terminal before, you might want the system manager to help you to turn on your terminal.

Turning on the Terminal

There are two types of terminals: video display terminals (called VDTs or CRTs) and hardcopy terminals. Video display terminals resemble television screens with attached keyboards, whereas hardcopy terminals look like electric typewriters threaded with large rolls of paper. We recommend that you use a video display terminal for the session because it is faster than a hardcopy terminal, and is capable of clearing the screen periodically.

A terminal is ready to use if it displays a message such as

```
**** system name  / Press NEW-LINE to begin logging on ****
```

If a terminal doesn't display this or a similar message, check to see that it's turned on. On recent video display terminals, the ON switch is a rocker switch on the back of the terminal. Press the switch toward ON, and wait 10 seconds or so for it to warm up. The terminal will display

```
Dxxx  Self Test  OK
```

Then press the NEW LINE key. In most cases, the terminal will display a banner with the following format:

```
AOS/VS nnn      / EXEC nnn      date   time   @CONn
Username:
```

Notice that a box or an underscore appears on the video display screen next to Username. This is called the *cursor* — it marks your place and shows where your keyboard input will appear.

In cases where you still have no response, check to see that the terminal is plugged in, and the ON LINE light, if any, is on. For more assistance, refer to the user's manual for the specific terminal.

Username and Password

As a valid user of an AOS/VS system, you have an account that's identified by two code words: a *username* and a *password*. The system manager selects these code words for you when he or she creates your account on the system. These codes authorize you to work on the system in your assigned area and use certain resources of the system.

For continuity in this book, we use the username *Alexis* and the password *defender*. However, your username and password will be different.

Although we have only one user in the session examples, in reality an AOS/VS system will have many of them. AOS/VS is capable of supporting hundreds of users simultaneously; on most AOS/VS sites, users will share system resources — unknowingly perhaps — with many people.

All valid users have a unique username/password code that admits them to the system. As members of a computing community, each user is responsible for protecting the system from any unauthorized intrusion. The best way to do this is by protecting your password: you should never reveal your password to anyone. Remember, only one slip, and the code loses its protective power.

Examining the Keyboard

Having turned on your terminal, look at its keyboard. Notice that the main keypad, shown in Figure 2-1, resembles that of a typewriter. The numbers 1 through 0 are at the top, alphabetic characters are below, and the space bar is at the bottom.

There are some extra keys on the main keypad. The most important are the BREAK/ESC (or just ESC) and the CTRL keys on the left, and the NEW LINE key to the right of the main keypad. The ESC and CTRL keys change the effect of regular keys, in ways that we'll discuss later in the session. The NEW LINE key directs the computer to take action; you'll be using it right away. (This manual frequently uses the symbol ⤸ to represent the NEW LINE key.) The CR (Carriage Return) key, which is next to the NEW LINE key, operates the same way as NEW LINE, *except* that it erases everything to the right of the cursor.

On the main keypad you'll also see the SHIFT key and ALPHA LOCK key. The SHIFT keys work the same way they do on a typewriter. The ALPHA LOCK key, if there is one, instructs the terminal to print all letters in uppercase. The ALPHA LOCK does not change numbers into the symbols: you must use the SHIFT key for that.

Note that the letter O and the number 0, as well as the letter l and the number 1 are *different characters*, although they look somewhat alike in typeset text.

To the *right* of the main keypad, you'll see two other keypads. The numeric keys on the far right can be used like a calculator for data entry. The pad between the main and numeric keypads is the cursor control keypad; it includes the HOME key and other directional keys. You'll use them in this chapter's practice session to reposition your cursor.

At the top of the keyboard, you'll see a row of mostly unmarked keys. These are called *function keys*, and they represent shorthand commands, specific to the program you're running. The session on the SED text editor (in Chapter 4) gives you practice with function keys.

Correcting Typing Errors

Whenever you make a typing error, you can rely on two or three keys to correct them. The DEL key in the upper right of the main keypad erases the character just to the left of the cursor. The DEL key in conjunction with the REPT key, below it, will erase many characters in a line. The space bar, moving to the right, will also erase a character, leaving a blank space in its place.

If by chance your screen appears frozen, and neither the NEW LINE key nor the DEL key seem to have an effect, press the CTRL key and the Q key simultaneouly. This will probably free the screen and you can continue with the session.

Logging On

Now that you're familiar with the keyboard, you're ready to start the session. To log on, you'll identify yourself as a valid user by typing your username and password. (To log on from a modem, see the next section.)

If the terminal displays a banner like this:

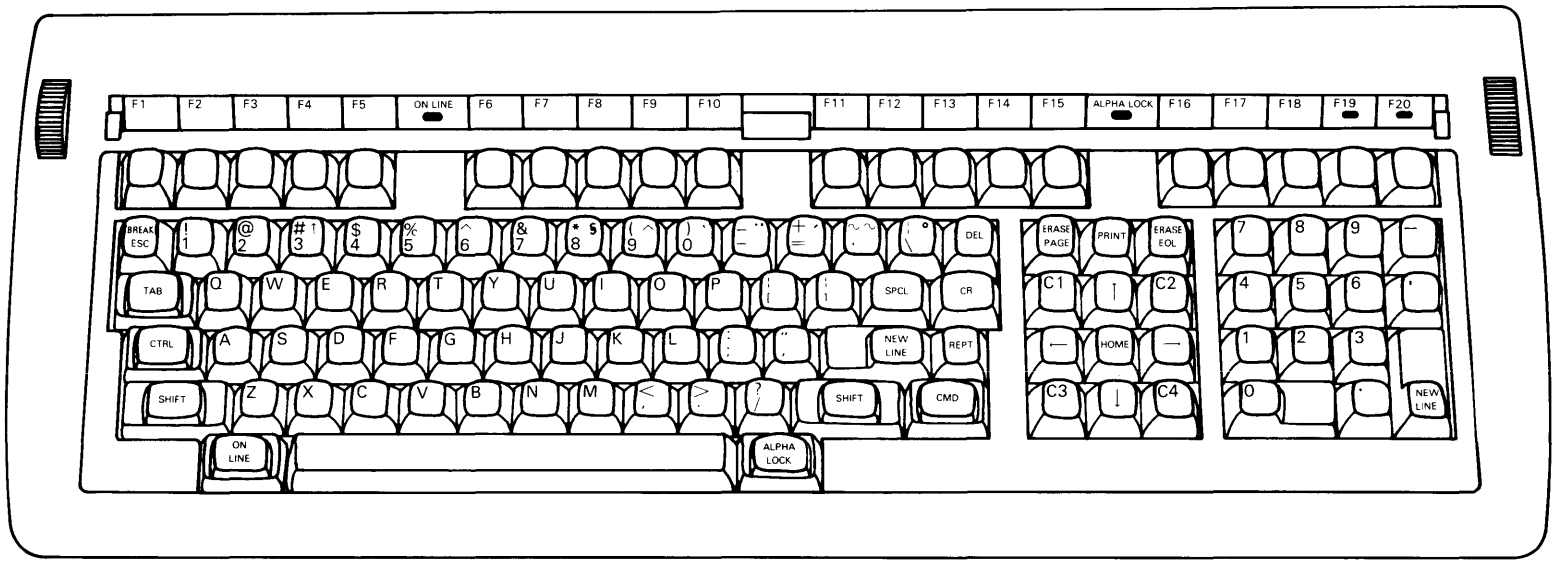
```
**** system name / Press NEW-LINE to begin logging on ****
```

Then press the NEW LINE key to signal the system that your're ready to log on:

```
⤸
```

At this point, everyone should see the AOS/VS banner:

```
AOS/VS n    /    EXEC    n    date    time    terminal-number
Username:
```



DG-25920

Figure 2-1. The DASHER® D211 Keyboard

The system prompts you for your username. Type in your username, and press the NEW LINE key. Take your time — you have 30 seconds or so to respond. Type your own username instead of Alexis, which is used throughout this session as the username:

```
Username:    Alexis )
```

The system also prompts for your password, but it won't display (echo) it for security reasons. Type in your password and press the NEW LINE key:

```
Password:            )
```

The system verifies your username/password code. If there is an inconsistency, it displays an error message:

```
Invalid username-password pair
Username:
```

It is possible to make a typing mistake and enter an incorrect username or password. Whenever you receive this message, simply enter your username and password again. (If you make several mistakes in entering the username/password combination, the system will display the message, *Too many attempts, console locking for 10 seconds*. No harm done. Wait 10 seconds and start again.

After you type your correct username/password pair, the terminal will display local system announcements, which are entered by the system manager, and the AOS/VS CLI banner:

... (Announcements) ...

```
Last previous logon    date    time

AOS/VS CLI    REV n     date    time
)
```

The right parenthesis that appears below the banner is the Command Line Interpreter (CLI) prompt. The CLI is the interactive monitor for AOS/VS; by using its commands, you gain access to all the programs and services available under AOS/VS.

Seeing the) prompt, you know you've logged on to the system. The CLI displays the prompt each time it completes a command: it signals you that the CLI is available to accept another command. You'll see it hundreds of times in the future.

Logging On Using a Modem

A modem makes it possible for you to log on a computer from a remote location. (A modem is a special device that transmits and receives signals over a communications line. When transmitting signals, it modulates digital sound to the necessary line frequency; when it receives signals, it demodulates (converts) received digital signals into their original frequency; thus its name, a *mo*dulator-*dem*odulator pair, or modem.) Those of you using a modem need to ask your system manager for more than a username/password code; you also need to know your computer's telephone number(s), what kind of answering service to expect, and how to set the modem switches for line speed and parity.

To log on a computer over a modem, first set up your terminal. Plug it in and turn the power on. At this time, adjust the line speed setting (it can range from 300 to 1200 baud — the modulation rate) to whatever the system manager said. Then dial the computer.

Either an operator or an automatic answering service (it's usually a high-frequency tone) will respond to your call. Complete the connection as described by your system manager. Then press the NEW LINE key, and proceed as explained in the preceding section, "Logging On." If you make several mistakes logging on, and the system displays the message, *Too many attempts, console locking for 10 seconds*, hang up, dial again, and re-enter your username/password code.

To log off, follow the directions in the following section. After logging off, simply hang up the phone and turn the modem off.

Logging Off

You can end an AOS/VS session at any time, by signing off the system with the BYE command. The BYE command directs AOS/VS to terminate your user process; it frees your terminal for anyone with a valid username and password to log on.

To log off, type BYE next to the CLI prompt and press the NEW LINE key. Try it, type

```
) bye ↵

AOS/VS  CLI    TERMINATING       date       time

Process  n  terminated
Connect time hours:minutes:seconds
User 'ALEXIS'  logged off @CONn    date     time
.
.
.
**** system name / Press NEW-LINE to begin logging on ****
```

If the terminal will be idle for a long time, such as overnight or over the weekend, turn the terminal off by pushing the switch near the lower right of the screen or by pressing the rocker switch on the back of the terminal toward off.

To log on again, turn on the terminal and press the NEW LINE key. Then type your username and password, as described earlier.

Changing Your Password

Your password and username give you access to the system. Your username is public knowledge and remains the same from one session to another. But your password is a secret code, known only to you. Never disclose your password to anyone. Only authorized users should gain access to AOS/VS, and if an unauthorized person wants to use the system for any reason, direct them to the system manager, rather than reveal your password.

Your password is what protects your files from public viewing and makes your account safe and secure. Always change your password as a new AOS/VS user so that you start out with a secret password. We also recommend that you change it from time to time to ensure continued privacy.

To change your password, log on as usual with your username and password. However, after you type the password, press the ERASE PAGE key instead of the NEW LINE key. The system will prompt for the new password, and you'll type it.

A password must be 6 to 15 characters long, and can be any combination of the following characters: upper- or lowercase letters, numbers 0 through 9, and all printing characters except the uparrow. Be sure to select a password that's fairly obscure.

A good password includes a mixture of letters, numbers, and symbolic characters and is longer than six characters. One way to build good, memorable passwords is to combine nonsense syllables like ang, fof, or lal. A password such as ANG..FOF is difficult for someone to guess.

For example, suppose Alexis has an initial password DEFENDER, which the system manager assigned. After learning to log on and off the system, Alexis chooses a truly private password such as SAN..TOR. To change DEFENDER to SAN..TOR, Alexis does the following:

Alexis starts logging on by pressing the NEW LINE key:

```
... Press NEW-LINE to begin logging on ...

)
```

AOS/VS displays its banner and prompts for username and password. Alexis types the usual username/password combination of Alexis and DEFENDER. However, instead of pressing the NEW LINE key after DEFENDER, Alexis presses the ERASE PAGE key:

```
... EXEC  n  date ...
Username:    Alexis )
Password:    defender (which doesn't echo)  (ERASE PAGE)
```

And AOS/VS prompts for the new password, which Alexis types

```
Enter your new password:     san..tor )
```

The system confirms the new password,

```
--New password in effect--
Last previous logon...
```

and logs Alexis on the system.

From this time on, Alexis's logon code is ALEXIS/SAN..TOR. The new password remains in effect until Alexis changes it again. (If connected by a network to other computers, Alexis must also change the old password to the new one on those systems.)

If you ever forget your password or username, see your system manager.

Summary

In this section you prepared a terminal for the upcoming session with AOS/VS; you noted the features of your computer keyboard and became familiar with username/password pairs. You now know how to log on and off the system, and how to change your password. You also have some idea of how to maintain the privacy of your account.

Getting Your Bearings

As mentioned earlier, the CLI is the command language (or interactive monitor) for AOS/VS. You use its commands to gain access to all the programs and services available under AOS/VS. Some CLI commands, such as the ones we look at in this section, orient you to AOS/VS. They provide you with such general information as the time of day, the date, and your location within the system.

Time

The TIME command displays the current system time. For example, type

```
) time ↵
11:24:16
```

The system time is 11:24:16 in the morning. Because AOS/VS uses a 24-hour clock, 1:30 in the afternoon is displayed as 13:30:00, and 10:07 at night as 22:07:18.

All CLI commands follow certain conventions. You can abbreviate any command to its shortest unique string, so instead of typing the full TIME command, you could have tried something shorter:

```
) t ↵
ERROR: COMMAND ABBREVIATION NOT UNIQUE
t
```

That was too short, but try

```
) ti ↵
11:25:01
```

The abbreivation "ti" is unique, so it's accepted. The CLI also allows either uppercase or lowercase letters — it's not case sensitive. It does require that you complete a command line by pressing the NEW LINE key or the CR key. (But remember that the CR key deletes everything to the right of the cursor.)

Date

The DATE command displays the current system date. Type

```
) date ↵
03-SEP-85
```

It's the third day of September 1985.

The CLI also allows you to stack commands on a line, separating them with semicolons. For example, you can type

```
) time;date ↵
11:37:04
03-SEP-85
```

AOS/VS displays the system time first, and then the date — just as you requested. You can even continue a command line over several lines as long as you end each line with the CLI continuation character, the ampersand (&). The CLI then prefaces its next prompt with an ampersand to signal a continued command line. For example, type

```
) ti& ↵
&  ) me;& ↵
&  ) da& ↵
&  ) te ↵
11:39:09
03-SEP-85
```

The CLI offers a lot of flexibility. Since it executes commands sequentially, it doesn't matter if commands span one line, four, or more.

User Directory

As a typical user of an AOS/VS system, you have your own directory, which we call your *user directory*. Here you create text files, and develop or run programs. The name of your directory is the same as your username, with a prefix of :UDD. The user directory in this session is :UDD:ALEXIS (see Figure 2-2).

One CLI command, the DIRECTORY command, displays your current directory within the system. For example, type

```
) dir ↓
:UDD:ALEXIS
```

Alexis works in an area known as :UDD:ALEXIS. The :UDD prefix means that Alexis is listed in directory UDD (User Directory Directory) as an authorized user of the system, and that UDD is subordinate to the root directory, symbolized by the colon (:). ALEXIS is the initial user directory for Alexis — Alexis always logs on in directory :UDD:ALEXIS.

You can see in Figure 2-2 that AOS/VS designates certain directories for its own use, and other directories for users of the system. The highest directory is called the root. Subordinate to the root are system directories for networking, peripheral devices (such as printers and tape units), and utilities (such as text editors and language compilers). Users of the system are listed in directory UDD, and their individual directories stem from it. Figure 2-2 shows three system users: Alexis, OP (the operator), and Chris.

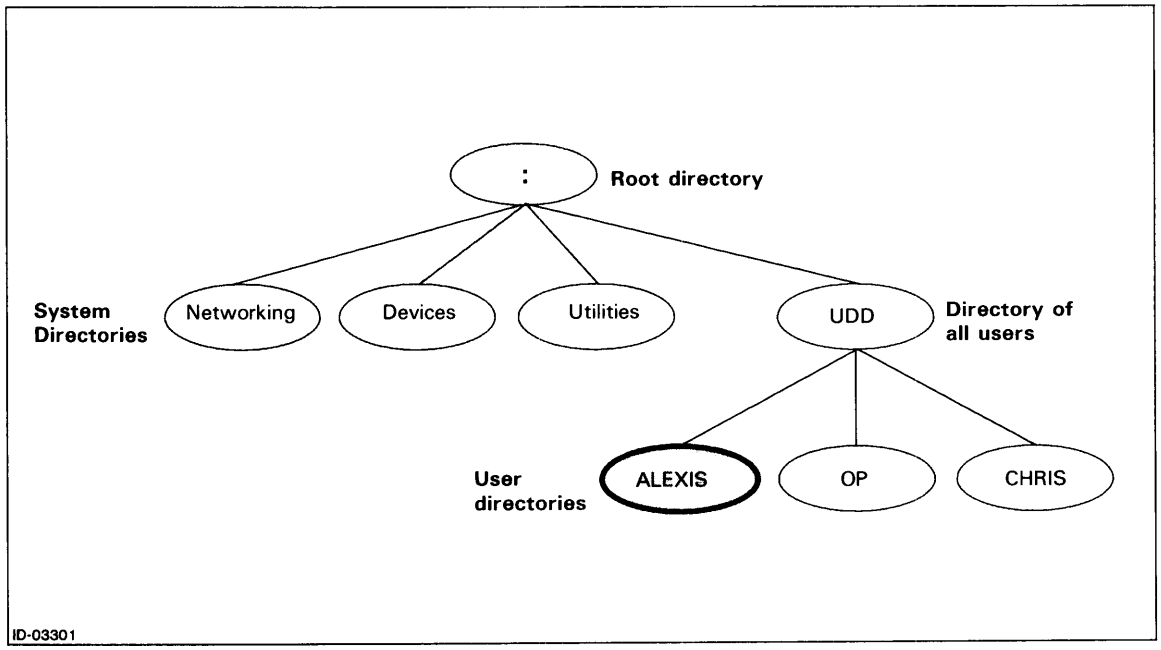


Figure 2-2. In Directory :UDD:ALEXIS

Process Identity

Whenever you log on AOS/VS, the CLI begins running a process for your terminal that you use to communicate with the system. From this initial CLI process you can run such other processes as the BASIC interpreter or a text editor. Each process that the CLI sets up for you has an identifying number, called a Process ID or PID.

The CLI WHO command displays information about the process you're running. For example, type

```
) who )
PID:  48   ALEXIS   CON3   :CLI.PR
```

The process ID for Alexis is number 48 — or PID 48, and the process is running a CLI program on CON3. CON3 is the terminal where Alexis logged on.

Summary

In this section of the AOS/VS session, we looked at CLI commands that provide general information about the system: TIME, DATE, DIRECTORY, and WHO. You'll find these commands helpful in your daily use of the system. Later on, when you're more skilled with the CLI, you can introduce these commands into CLI macros (command files), which we look at, among other kinds of files, in the next section.

Learning About Files

A file is any unit of information, any collection of data, that has a unique name. Files take many forms — they can be programs (sets of instructions), documents, commands, records of a computer session, personal letters, or directories that contain other files. AOS/VS stores files on disk in the same way you might store files in a file cabinet.

In this section we look at AOS/VS files, and you'll learn how to create and name them. You'll use CLI commands to display the names and contents of files. In addition, you'll work with two special kinds of files: CLI command files and log files.

Naming Files

An AOS/VS filename can be from 1 to 31 characters long, and contain any alphanumeric character (A–Z, a–z, 0–9) as well as an underscore (_), period (.), question mark (?), or dollar sign ($). This wide range of filename characters allows you to create explicit filenames. For example, the name of the file Alexis will write in this session is VS.PRACTICE. Other filenames could be FEB_85_EXPENDITURES or $FIRST.ANNUAL.FUNDRAISER — both clearly describe the file contents.

A suffix on a filename further defines its contents. For example, the .F77 suffix labels a FORTRAN 77 file, the .C suffix, a C program, the .PAS suffix, a Pascal program, and the .TXT suffix, a text file. While AOS/VS provides great latitude in the suffixes that you can assign, it does assume that .OB identifies an object file and that .PR defines a program file.

Creating Files

You create a file with the CLI CREATE command, a text editor, or a word processor. Ordinarily, you will use a text editor or the CEO word processor to make a new file, because both allow you to work with the file, inserting or changing the text or data. However, for simple files, the CLI CREATE command is adequate.

When using the CREATE command, supply the filename as an argument to the command. Remember that you can abbreviate CLI commands to the shortest, unique form. For example, type

```
) cre testfile ↵
```

You just created a file called TESTFILE. Note the format of the CREATE command: the filename TESTFILE is an *argument* to the command CREATE. The CLI requires that you insert at least one space between a command and its argument. You can use more than one space, a comma, one or more tabs, or any combination you like.

But TESTFILE is an empty file, so it isn't very useful.

To create a file and insert text, use the CREATE command with the /INSERT switch. The /I switch on the CREATE command modifies its meaning. It directs the CLI to create the file and then insert text from the keyboard into the file. During text insertion, the CLI displays the double prompt)) at the beginning of each new line. When the text is entered, signal the CLI by typing a right parenthesis (making three on the line) and press the NEW LINE key.

CREATE/I can be useful for writing little files; for example, type

```
) create/i vs.practice ↵
)) Greetings from the AOS/VS session. ↵
)) ) ↵
)
```

Now you have two files — the empty one, TESTFILE, and VS.PRACTICE with its greeting.

Viewing the Contents of a File

To look at the contents of a file, use the CLI TYPE command. The command, which can be abbreviated TY, displays the contents of a file on the terminal screen. Try it.

```
) ty vs.practice ↵
Greetings from the AOS/VS session.
```

The TYPE command works well with ASCII files, which are text files. Whenever you want to view non-ASCII files — those translated into machine language and identified by certain suffixes such as .PR — use the DISPLAY utility, described in the *Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS).*

Listing Files in a Directory

AOS/VS has cataloged TESTFILE and VS.PRACTICE. It's recorded that directory :UDD:ALEXIS contains two text files, both created on September 5.

To view the file catalog, use the FILESTATUS command; it can be abbreviated to F. Type

```
) f ↵

DIRECTORY :UDD:ALEXIS

  VS.PRACTICE  TESTFILE
```

The FILESTATUS command displays the working directory :UDD:ALEXIS (which serves as an address for the files) and verifies their existence.

If you include switches on the FILESTATUS command, it will provide even more information. The /ASSORTMENT switch directs the CLI to display an assortment of information about the file: the file type, the date and time created or modified, and its length in bytes. For example, type

```
) f/as ↓

DIRECTORY  :UDD:ALEXIS

  VS.PRACTICE          TXT   3-SEP-85  13:09:20      35
  TESTFILE             TXT   3-SEP-85  13:07:14       0
```

With the /AS switch you can see that both files are text files (TXT); one of them is 35 bytes (or 35 characters) long, including the NEW LINE character, and the other is empty. (Your file VS.PRACTICE might have a different number of bytes if you inserted extra spaces or characters. Obviously, the times and dates will be different for each of your files.)

All switches change the meaning of a command or the operation performed on its argument(s); they are a useful part of AOS/VS. There are *command switches* that attach to commands, and *argument switches* that attach to arguments.

Sometimes you might want information on certain files, not on the entire directory. You can name files as arguments to the FILESTATUS command. For example, to receive information about one file, VS.PRACTICE, type

```
) f/as vs.practice ↓

DIRECTORY :UDD:ALEXIS

  VS.PRACTICE   TXT      3-SEP-85     13:09:20   35
```

The system displays information on only the file you gave as an argument to the FILESTATUS command.

As your files multiply, you'll want an organized listing so it's easy to find what you need. The /SORT switch on the FILESTATUS command sorts the files into alphabetical order. Type

```
) f/as/s ↓

DIRECTORY  :UDD:ALEXIS

  TESTFILE             TXT   3-SEP-85  13:07:14       0
  VS.PRACTICE          TXT   3-SEP-85  13:09:20      35
```

Alexis's directory doesn't provide much of a challenge for the sort switch (/S), but the example makes the point. The files appear in alphabetical order.

Using Templates to Find Files

Even with an alphabetic listing of files, you'll find it time-consuming to scan for required files. After a while your directories will hold a large number of files. Instead of looking at an entire listing, it's possible to look at a certain subset of files — for example, FORTRAN 77 files, which all end in .F77.

The CLI recognizes a number of symbols called templates that you use to represent parts of filenames in various ways. Template characters — which can be *, –, +, \, # — have the following meanings:

Template Character	Meaning
* (asterisk)	Matches any single character except a period.
– (hyphen)	Matches any series of characters not containing a period.
+ (plus sign)	Matches any series of characters.
\ (backslash)	*Omits* a series of characters.
# (number sign)	Matches the file(s) in the specified directory *and all subordinate directories.*

A template either stands alone or is used with other characters and templates. For example, to see all filenames eight characters long that don't have periods, you can enter the FILESTATUS command with eight asterisks as the argument. Type

```
) f ******** ↵

DIRECTORY  :UDD:ALEXIS

  TESTFILE
```

TESTFILE has eight characters but no periods.

To see filenames of any length that don't contain a period, use the hyphen. For example, type

```
) f - ↵

DIRECTORY :UDD:ALEXIS

  TESTFILE
```

Again, TESTFILE meets the template requirements as a filename of any length, without periods. To search, regardless of periods, use the plus sign. Type

```
) f + ↵

DIRECTORY  :UDD:ALEXIS

  VS.PRACTICE      TESTFILE
```

You'll frequently use the plus sign and a suffix to obtain a listing of a file group, such as all C source files or all Pascal source file. Assume you have many FORTRAN 77 source files, which conventionally end in .F77. You want to see all their names sorted. Type

```
) f/as/s +.f77  ↵
```

The resulting listing might be

```
ARRAY.F77          UDF    10-JAN-85    10:46:50      2297
BIORHYTHM.F77      UDF    12-JAN-85     9:01:56      2337
MASK.F77           UDF    10-JAN-85    14:48:01      8334
MORTGAGE.F77       UDF     9-JAN-85    11:01:33      2023
OPEN_TEST.F77      UDF     9-JAN-85    13:14:02      1356
```

As you can see, a template acts as a filter — it allows some filenames through and it rejects others.

Other template characters, the backslash (\) and the number sign (#) come in handy in printing or reorganizing files. We'll use them later in the session.

Writing Macros (Command Files)

Now let's look at an AOS/VS command file — called a macro. Under AOS/VS you can create a file containing a series of CLI commands, and then direct the CLI to execute the contents of the file.

To create a macro, use the CLI CREATE command and attach a .CLI suffix to the filename to distinguish it as a command file. Then enter a series of CLI commands, in the order you want them executed. To execute the macro, type the filename without the suffix. When AOS/VS sees the .CLI ending, it knows to execute the contents of the file.

The CLI WRITE command is an aid when writing macros. It directs the CLI to print or display a text string that's included as an argument to the command. (In the following example, VS.PRACTICE serves as an argument to the WRITE command.)

To create a macro (which we will call MACRO.CLI for emphasis) type the following characters, separate the commands with a semicolon, and then close the file with a right parenthesis and NEW LINE:

```
) cre/i macro.cli ↵
)) WRITE VS.PRACTICE contains ; type vs.practice  ↵
)) ) ↵
```

Now, when you type the macro name, the CLI will execute both the WRITE and TYPE command lines in the file. Try it:

```
) macro ↵
VS.PRACTICE contains
Greetings from the AOS/VS session.
```

And it did! You'll write many macros in your work under AOS/VS. You'll find them a powerful feature of AOS/VS.

Now, having created three files, let's check them with the FILESTATUS command. But first, test the error checking of AOS/VS; type

```
) fiels/as ↵
ERROR: NOT A COMMAND OR MACRO, fiels/as
fiels/as
```

When you make a mistake, AOS/VS displays the faulty line and comments on the error. Sometimes the comment is specific and other times, it seems vague. But AOS/VS won't log you off or interrupt your computing session.

Re-enter the command, this time in its abbreviated form:

```
) f/as ↵

DIRECTORY   :UDD:ALEXIS

  TESTFILE          TXT   3-SEP-85  13:07:14      0
  MACRO.CLI         TXT   3-SEP-85  13:25:50     47
  VS.PRACTICE       TXT   3-SEP-85  13:09:20     35
```

The new command file, MACRO.CLI appears in the listing, and it's 47 bytes long.

Now that we have a macro, let's use some template characters in arguments to the FILESTATUS command. To list all macros in a directory, type

```
) files  -.cli  ↵

DIRECTORY  :UDD:ALEXIS

  MACRO.CLI
```

To see all filenames, including periods, that begin with M, type

```
) files m+  ↵

DIRECTORY  :UDD:ALEXIS

  MACRO.CLI
```

To see all filenames that do *not* end in .CLI, type

```
) files  +\+.cli ↵

DIRECTORY  :UDD:ALEXIS

  VS.PRACTICE TESTFILE
```

And the system displays all files (+) except those that end in .CLI (\+.CLI).

Record Keeping with a Log File

Just as sailors record navigational events in a ship's log, the CLI will record your computing session in a file called a log file. The log file reports all CLI dialog directed to or from your terminal. For example, the following log file (Figure 2-3) reports our AOS/VS session to this point.

Examining the log file, you'll notice that certain parts of the dialog, such as displays resulting from the TYPE command, are not included. Nevertheless, the log file is a useful record of your session with the CLI.

```
) time
13:53:43
) t
ERROR: COMMAND ABBREVIATION NOT UNIQUE
t
) ti
13:53:48
) date
 9-SEP-85
) time;date
13:53:56
 9-SEP-85
) ti&
&)me;&
&)da&
&)te
13:54:08
 9-SEP-85
) dir
:UDD:ALEXIS
) who
PID:    62 ALEXIS          CON11          :CLI.PR
) cre testfile
) create/i vs.practice
))Greetings from the AOS/VS session.
)))
) ty vs.practice

) f

DIRECTORY :UDD:ALEXIS

  VS.PRACTICE       TESTFILE
```

Figure 2-3. A Sample Log File (continues)

```
) f/as

DIRECTORY :UDD:ALEXIS

  VS.PRACTICE         TXT   9-SEP-85  13:55:48      35
  TESTFILE            TXT   9-SEP-85  13:55:36       0
) f/as vs.practice

DIRECTORY :UDD:ALEXIS

  VS.PRACTICE         TXT   9-SEP-85  13:55:48      35
) f/as/s

DIRECTORY :UDD:ALEXIS

  TESTFILE            TXT   9-SEP-85  13:55:36       0
  VS.PRACTICE         TXT   9-SEP-85  13:55:48      35
) f ********

DIRECTORY :UDD:ALEXIS

  TESTFILE
) f -

DIRECTORY :UDD:ALEXIS

  TESTFILE
) f +

DIRECTORY :UDD:ALEXIS

  VS.PRACTICE       TESTFILE
) f/as/s +.f77
) cre/i macro.cli
))write VS.PRACTICE contains  ; type vs.practice
)))
) macro
VS.PRACTICE contains

) fiels/as
ERROR: NOT A COMMAND OR MACRO, fiels/as
fiels/as
) f/as

DIRECTORY :UDD:ALEXIS

  MACRO.CLI           TXT   9-SEP-85  13:57:14      47
  VS.PRACTICE         TXT   9-SEP-85  13:55:48      35
  TESTFILE            TXT   9-SEP-85  13:55:36       0
```

Figure 2-3. A Sample Log File (continued)

```
) files -.cli

DIRECTORY :UDD:ALEXIS

  MACRO.CLI
) files m+

DIRECTORY :UDD:ALEXIS

  MACRO.CLI
) files +\+.cli

DIRECTORY :UDD:ALEXIS

  VS.PRACTICE       TESTFILE
```

Figure 2-3. A Sample Log File (concluded)

To create your own log file, use the LOGFILE command and supply a name. Type

```
) logfile session.log ↓
```

The file SESSION.LOG will grow during the AOS/VS session, and provide a good summary. You can close the log file using the /KILL switch, by typing

```
) logfile/k ↓
```

or by logging off the system.

When you log on again, you have to open the log file with the LOGFILE command. The system appends new material to the original file, if the original filename is the argument, or it starts a new file if you supply a new filename argument. Each log file remains in the directory where you started it, but you can have only one log file open at a time. At the end of the session, you can print the log file or examine it with the TYPE command. Because the log file can become pretty large (check its size with the FILESTATUS command), you might want to delete it with the DELETE command, which we discuss later in the session.

Summary

Now you know how to create and name a file with the CREATE command, and display its contents with the TYPE command. Furthermore, you can see a listing of files within a directory using the FILESTATUS command, build a simple macro, and create and terminate a log file.

Working with Directories

A directory is a catalog of files. In a way, it resembles the drawers of a file cabinet. The primary reason for directories is to let you keep similar files together. For example, you could put all FORTRAN programs in one directory, all COBOL programs in another, all macros in yet another, all personal notes or memos in another, and monthly expense sheets in another. As many "topics" as you have, you can have a directory of files for each.

As a user, you have your own *user directory*, with the username you logged on with. Your user directory is the exclusive property of your username — in this session, the property of Alexis. From your initial user directory you can create subordinate directories, and move to those directories when you want to perform some job. For example, you can create a BASIC directory and make it your working directory when writing BASIC programs. The directory in which you're located is always called your *working directory*.

This section of the AOS/VS session shows how to create directories and move from one to another. We also discuss how to use directories outside your work area.

Creating a Directory

Creating directories is easy: you just use the CREATE command with the /DIRECTORY switch and specify the new directory name as an argument. Directory names follow the same naming conventions as regular files. For example, to create a subordinate directory to :UDD:ALEXIS called LEARNING, type

```
) create/dir learning ↵
```

Use the FILESTATUS command to confirm the existence of directory LEARNING. Type

```
) f/as/s ↵

DIRECTORY  :UDD:ALEXIS

  LEARNING            DIR   3-SEP-85  13:38:30       0
  MACRO.CLI           TXT   3-SEP-85  13:25:50      47
  SESSION.LOG         TXT   3-SEP-85  13:14:12     341
  TESTFILE            TXT   3-SEP-85  13:07:14       0
  VS.PRACTICE         TXT   3-SEP-85  13:09:20      35
```

The new directory LEARNING appears in the file listing as a directory file, type DIR.

Sometimes you will want to move files from one directory to another, so let's create a second directory and call it RECORDS. It will also be subordinate to :UDD:ALEXIS. Type

```
) cre/dir records  ↵
```

Now check the file listing to verify the new directory:

```
) f/as/s records  ↵

DIRECTORY  :UDD:ALEXIS

  RECORDS             DIR   3-SEP-85  13:39:54       0
```

Now Alexis has three directories, as seen in Figure 2-4. The initial and current directory, called ALEXIS, is the working directory. LEARNING and RECORDS are subordinate to it. Both subordinate directories are currently empty.

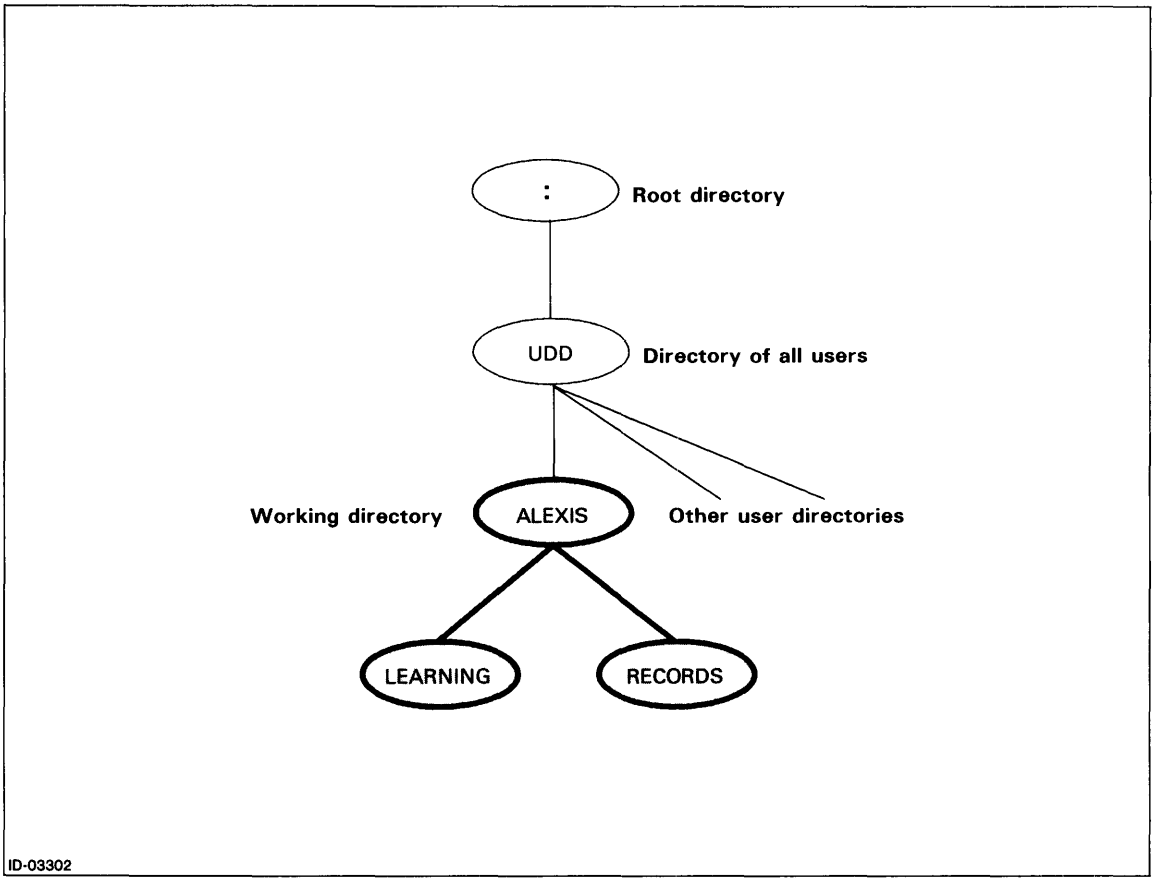


Figure 2-4. Directory Structure of :UDD:ALEXIS

Changing Directories

Your personal workspace in AOS/VS extends from your initial user directory (in this session :UDD:ALEXIS) down to your most subordinate directory. AOS/VS allows many levels of directories beneath your initial user directory. As you work, you can change your location from your initial directory to the directory where you need to be.

To move from one directory to another, you'll use the DIRECTORY command with the new directory-name as an argument. For example, to change to directory LEARNING, type

```
) dir learning ↵
```

Now, to verify that the working directory is LEARNING, issue the DIRECTORY command without an argument. Type

```
) dir ↵
:UDD:ALEXIS:LEARNING
```

The system confirms that your new location is directory LEARNING.

Pathnames

:UDD:ALEXIS:LEARNING describes the complete path to the LEARNING directory from the root directory. It's called a *pathname*.

Figure 1-3 is an illustration of the AOS/VS directory structure. If you return to it, you'll see that the AOS/VS structure resembles an inverted tree. At the top of the tree is the root directory. You can trace a route through the branches of the tree by taking each name in the pathname as a new branch.

Figure 2-4 focuses on the directories in the pathname :UDD:ALEXIS:LEARNING. You'll notice that the path begins at the root directory (:). (The first colon in all pathnames refers to the root directory; thereafter, colons act as separators.) Directory UDD, which lists all system users, is directly below the root. Below UDD there's Alexis's initial user directory, ALEXIS, and its subdirectory LEARNING.

A pathname can refer to a directory or a file within a directory. For example, :UDD:ALEXIS is a pathname to the directory ALEXIS, whereas :UDD:ALEXIS:VS.PRACTICE is a pathname to the file VS.PRACTICE within the directory ALEXIS.

Note that a colon separates a directory name from a filename. All names in a pathname must be separated by colons. A space is a delimiter (it defines the end of the text string) and cannot be part of a filename or pathname.

Referring to Directories

You'll also use a pathname when you reorganize files, moving them into different directories. To relocate files, use the CLI MOVE command, followed by the pathname of the destination directory, and the files you want to move. You must be *in the directory* that holds the files you want to move.

For example, to relocate files from :UDD:ALEXIS to :UDD:ALEXIS:LEARNING, first return to the directory that holds the files, :UDD:ALEXIS. Type

```
) dir :udd:alexis ↓
```

Confirm that the files are there by typing

```
) f/as/s ↓

DIRECTORY  :UDD:ALEXIS

  LEARNING          DIR   3-SEP-85  13:38:30      0
  MACRO.CLI         TXT   3-SEP-85  13:25:50     47
  RECORDS           DIR   3-SEP-85  13:39:54      0
  SESSION.LOG       TXT   3-SEP-85  13:14:12    647
  TESTFILE          TXT   3-SEP-85  13:07:14      0
  VS.PRACTICE       TXT   3-SEP-85  13:09:20     35
```

Move MACRO.CLI and VS.PRACTICE into directory LEARNING with the CLI MOVE command and /VERIFY (/V) switch. The /V switch on the MOVE command requests the CLI to verify the files that it moves. Include LEARNING (the destination directory) as the first argument and the filenames you want to move as subsequent arguments. Type

```
) move/v  learning  macro.cli  vs.practice ↓
MACRO.CLI
VS.PRACTICE
```

Now a copy of the files exists in :UDD:ALEXIS, as well as in LEARNING. Since you don't need two copies of a file, eliminate the files from UDD:ALEXIS with the DELETE command and /VERIFY switch:

```
) delete/v  macro.cli  vs.practice ↓

DELETED macro.cli
DELETED vs.practice
```

Now change your working directory to directory LEARNING and display the files with the FILESTATUS command. Type

```
) dir learning ↓

) f  ↓

DIRECTORY  :UDD:ALEXIS:LEARNING

  MACRO.CLI    VS.PRACTICE
```

Because the LEARNING directory is immediately subordinate to :UDD:ALEXIS it wasn't necessary to use a long pathname in the command line, just the directory name. However, to move the two files, MACRO.CLI and VS.PRACTICE, from LEARNING into directory RECORDS requires a longer pathname. It's necessary to go through the superior directory, :UDD:ALEXIS in order to get to RECORDS. To try it, type

```
) move/v  :udd:Alexis:records  macro.cli  vs.practice ↓
MACRO.CLI
VS.PRACTICE
```

Now change your working directory to RECORDS and request a file listing. Type

```
) dir :udd:Alexis:records  ↓

) f/as/s ↓

DIRECTORY  :UDD:ALEXIS:RECORDS

  MACRO.CLI  ...
  VS.PRACTICE  ...
```

Both subdirectories now contain the files MACRO.CLI and VS.PRACTICE. Each file is the same, but their full names are different because a filename also includes its complete pathname. For example, one macro is known as :UDD:ALEXIS:RECORDS:MACRO.CLI, but the other is :UDD:ALEXIS:LEARNING:MACRO.CLI.

Accessing Files

In order to work with a file that's not in your current directory, you need to change your working directory or use a pathname to the file. For example, try to display the log file you created earlier. Type

```
) ty session.log ↓
WARNING: FILE DOES NOT EXIST, FILE session.log
```

SESSION.LOG does exist. But it's in the initial user directory! You either need to leave directory RECORDS, or provide the full path to the file. Try the complete pathname:

```
) ty :udd:Alexis:session.log ↵

) time
13:53:43
) t
ERROR: COMMAND ...
```

This time you were successful because the pathname :UDD:ALEXIS:SESSION.LOG provides a complete route to the file.

Listing Directories

The # template character, coupled with the FILESTATUS command, allows you to search multiple directories for a file. It expands to + and +:+ and +:+:+ and so on, until all subordinate directories have been matched. In this case you have two levels of directories, so it expands to + and +:+, matching ALEXIS, and subdirectories ALEXIS:LEARNING and ALEXIS:RECORDS. For example, to search all directories for the file named MACRO.CLI, type

```
) f/as :udd:Alexis:#:macro.cli ↵

DIRECTORY    :UDD:ALEXIS:LEARNING

    MACRO.CLI     TXT    date    time   nnnn

DIRECTORY    :UDD:ALEXIS:RECORDS

     MACRO.CLI     TXT    date    time   nnnn
```

And the system confirms that two directories have copies of MACRO.CLI.

To receive a listing of all files in and subordinate to :UDD:ALEXIS, use the number sign alone. Type

```
) f/as/s  :udd:Alexis:# ↵

DIRECTORY  :UDD

  ALEXIS        CPD     ...

DIRECTORY    :UDD:ALEXIS ...

  LEARNING      DIR  ...
  RECORDS       DIR  ...
  SESSION.LOG   TXT  ...
  TESTFILE      TXT  ...

DIRECTORY    :UDD:ALEXIS:LEARNING

  MACRO.CLI     TXT  ...
  VS.PRACTICE   TXT  ...

DIRECTORY    :UDD:ALEXIS:RECORDS

  MACRO.CLI     TXT  ...
  VS.PRACTICE   TXT  ...
```

Shortcuts in Changing Directories

Ordinarily you will use the DIRECTORY command and a pathname to change directories. For example, to change from RECORDS to LEARNING, you need to provide a full pathname from the root directory. Type

```
) dir :udd:Alexis:learning ⤸
```

Now confirm that LEARNING is your working directory. Type

```
) dir ⤸
```

```
DIRECTORY   :UDD:ALEXIS:LEARNING
```

It's easy to see that pathnames become cumbersome. So we use a shorthand character — the caret (^) — to change directories. (To type the caret, press the SHIFT key, and while depressing it, also type a 6.) A caret in conjunction with the DIRECTORY command makes the CLI go up one directory, and move to the superior directory. Try it, type

```
) dir ^ ⤸
```

```
) dir ⤸
```

```
DIRECTORY  :UDD:ALEXIS
```

The caret (^) is also useful for moving to parallel directories. To move from directory LEARNING to directory RECORDS doesn't require a full pathname. For example, return to directory LEARNING:

```
) dir learning  ⤸
```

Now to move from LEARNING to ALEXIS, and then down to RECORDS (make sure there's no space between the caret and the directory name), type

```
) dir  ^records  ⤸
```

And now confirm. Type

```
) dir ⤸
```

```
DIRECTORY  :UDD:ALEXIS:RECORDS
```

A full pathname always works, but can be laborious to type — so use the caret (^) as needed. You can use several carets in a pathname, each caret representing a directory level higher than your current location.

Another shortcut in changing directories is the /I switch. It tells the CLI to return you to your intial working directory. DIR/I, when issued from a directory outside your user directory, always returns you to your user directory. For example, relocate to directory :UTIL, and return with the DIR/I command. Type

```
) dir :util ↓

) dir ↓
:UTIL

) dir/i ↓

) dir

DIRECTORY :UDD:ALEXIS
```

In most cases, as you work with AOS/VS, you'll find it easier to relocate with the DIRECTORY command and carets or the /I switch, rather than use long pathnames.

Summary

In this section we looked at directories and the AOS/VS directory structure. You created two directories, used the FILESTATUS command to list different directories, and worked with paths and pathnames. Finally, you learned a few shortcuts to save you keystrokes and time.

Special Keys Under AOS/VS

AOS/VS has special control key combinations that let you control the screen display and save time. The CTRL key coupled with an ordinary key — such as A, B, C — produces a special effect. In some cases, a key combination moves the cursor quickly across a line, permitting you to change or insert text, or to open the text line so you can insert words. In other cases, a CTRL-key combination will clear the screen display or cancel a CLI command.

In this section we discuss two types of keys: screen edit keys, used to reposition the cursor, and system control sequences, used to control system output to the terminal screen.

Screen Editing Keys

The CTRL key when joined with such characters as A, E, or F produces a simple screen editor, which can save you a lot of aggravation when you make a typing mistake or want to re-enter a long command line. Although a description of screen editing may seem long and involved, the actual task is really very easy and fast.

These CTRL characters work the same way with both the SED text editor and the CLI, so you may be using them often. Don't be afraid to experiment with them.

The major screen editing characters and their effects are as follows:

Key Combinations	Effect
CTRL-A	Redisplays the last command typed.
CTRL-F	Moves the cursor forward to the next word, or redisplays the next word of the last command typed.

Key Combinations	Effect
CTRL-B	Moves the cursor backward to the previous word.
→ key or CTRL-X	Moves the cursor forward one character.
← key or CTRL-Y	Moves the cursor backward one character.
CR or ERASE EOL	Deletes all characters to the right of the cursor. CR also directs the CLI to execute the command.
CTRL-E	Opens the line for editing, allowing insertion of text; or closes it.
HOME or CTRL-H	Returns the cursor to the beginning of the line.

Now let's try a little screen editing. (Skip this section if you're working on a printing console.) First confirm that you're in your initial directory by typing

```
) dir ↵
DIRECTORY  :UDD:ALEXIS
```

Now, with the ring finger (or little finger) on your left hand, press the CTRL key and hold it down. While you're holding CTRL down, press the A key. (This combination of keys is called CTRL-A.) Release both keys. On your screen, you will see the last command line repeated, just as you originally typed it:

```
) dir
```

Look at the screen cursor, which is either a box or an underscore, depending on your terminal. The cursor will be at the end of the line, after DIR. Type a space followed by RECORDS and press the NEW LINE key:

```
) dir  records ↵

DIRECTORY  :UDD:ALEXIS:RECORDS
```

By pressing NEW LINE, you re-entered the DIRECTORY command, the last command you typed.

Now let's try entering a command line that we used before — one that will be an error. Type

```
) move/v  ^learning  macro.cli  vs.practice ↵

WARNING: FILE NAME ALREADY EXISTS: FILE ^learning:MACRO.CLI
WARNING: FILE NAME ALREADY EXISTS: FILE ^learning:VS.PRACTICE
```

The MOVE command provoked an error message because the files already exist in directory LEARNING. (Each file in a directory has to be unique. If you want to keep multiple versions of a file, add a number or some other identifier to the filename.) The error is okay. It was deliberate and no harm was done. Enter CTRL-A again, and the MOVE command will appear once more:

```
) move/v  ^learning  macro.cli  vs.practice
```

Now you will use screen edit keys to change the MOVE command to a DELETE command (because it wastes disk space to have multiple copies of a file). The DELETE command will eliminate the files from your working directory, in this case :UDD:ALEXIS:RECORDS. Copies of the files will remain in directory LEARNING.

Let's edit the command line on the screen. The cursor is at the end of the line. Press the HOME key at the center of the cursor control keypad. (If there is no HOME key, press CTRL-H.) HOME (or CTRL-H) moves the cursor to the beginning of the line, after the CLI prompt.

Now, type DELETE/V. This overwrites the MOVE/V portion of the command. The command should now look like this:

```
) delete/v learning   macro.cli   vs.practice
```

The cursor will be ahead of the last character typed, in this case on the l after the v. Now press CTRL-E to open up the line, giving you space between the v and the l. Type in /C (for /CONFIRM), and press CTRL-E again to close the line. Press the space bar to blank out the rest of LEARNING. The command line should look like this:

```
) delete/v/c          macro.cli   vs.practice
```

The cursor should be before MACRO.CLI. Now enter CTRL-F to move the cursor forward to the next word. Keep entering CTRL-F until the cursor is at the end of the command line.

With the cursor at the end of the line, press NEW LINE, directing the CLI to execute the DELETE command.

Since you included the /C switch, the system waits for confirmation before proceeding. Respond to each query with a Y and NEW LINE:

```
macro.cli?    y ↵
DELETED macro.cli
vs.practice?    y ↵
DELETED vs.practice
```

Now MACRO.CLI and VS.PRACTICE reside only in directory LEARNING.

At this point in the section, you've used some of the screen edit characters to display and modify a command line. CTRl-A redisplays a line; CTRL-F moves the cursor forward one word. CTRL-E opens and closes a line, allowing text insertion. Other keys available to you are: CTRL-B, which moves the cursor back one word, and CR, which deletes everything to the right of the cursor and enters the remaining command.

Screen Display Keys (or System Control Sequences)

The CTRL key used in combination with such characters as S, Q, and L regulates your screen display. Some screen control sequences can clear the screen of information, or stop and restart system output (so the lines don't go whizzing by). Other CTRL sequences will cancel a command or an active process.

CTRL-S and CTRL-Q are useful when you display a long file with the TYPE command. Since the CLI usually displays a file more quickly than you can read it, you can use CTRL-S to freeze the screen display, thus breaking the file into readable segments. To resume the display and view more of the file, enter CTRL-Q.

The most common system control sequences are as follows.

CTRL Sequence	What It Does
CTRL-L	Clears the screen, and executes the command next to the) prompt, if any.
CTRL-S	Freezes the screen display. To restart the suspended display, enter CTRL-Q.
CTRL-Q	Restarts the display suspended by CTRL-S.
CTRL-O	Discards the display of output to the screen so programs will run faster. To restart the screen display, enter CTRL-O again.
CTRL-U	Erases the current CLI command line.
CTRL-C CTRL-A	Interrupts execution of the current command. Effective in the CLI, the SPEED and SED text editors, and several languages.
CTRL-C CTRL-B	Stops the current program, and returns control to its parent process. If entered from the CLI, CTRL-C CTRL-B logs you off the system.

Let's try some of these sequences.

To clear the screen of old information, enter CTRL-L. (Hold down the CTRL key and type L.) You'll see your screen cleared, ready for the exercise.

Let's try the CTRL-S CTRL-Q sequence with a file in the directory :UTIL called PARU.32.SR. Type

```
) ty :util:paru.32.sr ↵
```

```
COPYRIGHT (C) DATA GENERAL CORPORATION ...
```

Now press down the CTRL key and type S.

```
CTRL-S
```

As you see, the display stops. To continue from where it stopped, enter

```
CTRL-Q
```

```
.DUSR  ERFDE= 25   ; FILE DOES...
.DUSR  ERNAE= 26   ; FILE ALREADY...
.DUSR  ERNAD= 27   ; NON-DIRECTORY...
.DUSR  EREOF= 30   ; END OF FILE.
.
.
.
```

To freeze the display again, enter

```
CTRL-S
```

However, as you probably don't want to read the rest of PARU.32.SR, or wait the several minutes needed to display it all on your terminal, you can cancel the TYPE command with the CTRL-C CTRL-A sequence. Press the CTRL key and type C; then, while depressing the CTRL key, press A. Enter

```
CTRL-C CTRL-A
```

CTRL-C CTRL-A should interrupt the TYPE command, but it appeared to do nothing. The control sequence *did*, in fact, interrupt the TYPE command, but you can't tell because display is still suspended with CTRL-S. So enter CTRL-Q to resume the display:

```
CTRL-Q
ERROR: CONSOLE INTERRUPT
```

A console interrupt isn't really an error, but this is the message the CLI displays when it cancels a command. Now practice the CTRL-S CTRL-Q combination. To repeat the command line, enter CTRL-A. The TYPE command line reappears (TY :UTIL:PARU.32.SR). Press NEW LINE, and alternately press CTRL-S and CTRL-Q to freeze and unfreeze the display. Finally, when you're bored by PARU.32.SR, enter CTRL-C CTRL-A to cancel the display.

From this exercise, you'll note that when CTRL-S is in effect, the system appears dead. It isn't — all you need do to receive system response is enter CTRL-Q.

Another sequence, CTRL-O, produces the same visual effect as CTRL-S, but it actually directs the CLI to discard output rather than display it on the screen. CTRL-O can help speed up programs that would otherwise display a lot of writing on the screen, because data transmission to terminals slows programs down. You'll rarely want to use CTRL-O when you're in the CLI. However, if the system appears frozen and CTRL-Q has no effect, try typing CTRL-O. CTRL-O also restarts a display.

CTRL-U can be handy when you've typed a long CLI command line, want to erase it, and don't want to press DEL many times. Enter

```
) A long, erroneous CLI command.        (CTRL-U)
)
```

CTRL-U erases the line.

Having tried CTRL-L, CTRL-S CTRL-Q, CTRL-C CTRL-A, and CTRL-U, all that remains is CTRL-C CTRL-B — the most powerful control sequence of all.

CTRL-C CTRL-B stops (aborts) a process. It's most useful during the compilation of programs when you discover that there are mistakes in the source program, and you don't want to wait for the compiler to finish. Be careful using this sequence — if you enter CTRL-C CTRL-B from a text editor, all your work will vanish; if you issue it from the CLI, you'll be logged off the system and have to log on again.

Don't type the sequence now because it will log you off the system. To try CTRL-C CTRL-B, start another process, such as the SED text editor. Type

```
) xeq sed )
Name of file to edit:
```

Instead of answering the question, enter CTRL-C CTRL-B.

```
CTRL-C CTRL-B
```

The text editor, SED, will display a message indicating that a terminal issued a termination request, and the program stops running:

```
*ABORT*
CONSOLE INTERRUPT
ERROR: FROM PROGRAM
xeq,sed
```

As you can see, CTRL-C CTRL-B has a mighty effect. Use it carefully.

Summary

You've learned the special keys of AOS/VS. You've experimented with the CTRL key and seen how different sequences reposition the cursor, allowing you to correct or add to command lines or text.

The system control sequences (the CTRL key paired with the keys A, C, B, L, and U) help you read text files and Help messages. They can speed up programs with intensive input/output or stop a program.

More About Files and Directories

Over time you'll acquire many files and directories, and the ways in which you use them will change. A number of CLI commands allow you to reorganize your directory and nondirectory files, bringing them up to date.

Moving Files

In an earlier section, you used the MOVE command to copy files from one directory to another. You moved copies of MACRO.CLI and VS.PRACTICE into two directories subordinate to :UDD:ALEXIS. Later on, you erased these files from the user directory :UDD:ALEXIS and, still later, from :UDD:ALEXIS:RECORDS.

Now you're going to use the MOVE command a different way. Assume you want to take the files in directory LEARNING and move them back to the initial user directory :UDD:ALEXIS. To do this, change your directory to LEARNING, since you need to be in the same directory as the files you want to move.

```
) dir :udd:Alexis:learning ⤸
```

To relocate the contents of LEARNING, issue a MOVE command with a destination directory of :UDD:ALEXIS. Rather than typing :UDD:ALEXIS, simply use the caret (^) to indicate you want the files moved to the superior directory. Rather than naming files, omit an argument. This tells the system to move all files in the working directory. Type

```
) move/v  ^   ⤸
MACRO.CLI
VS.PRACTICE
```

To summarize the symbols on the command line: the caret symbolizes the immediately superior directory, :UDD:ALEXIS; the number sign (#) means all files; the /V switch requests that the CLI verify the files it moved.

The original files are now in directory :UDD:ALEXIS. You can delete the files from directory LEARNING, where you are, by typing

```
) del/v/c  +  ⤸
=MACRO.CLI?     y ⤸
DELETED =MACRO.CLI
=VS.PRACTICE?     y ⤸
DELETED =VS.PRACTICE
```

Return to your initial directory with the command line

```
) dir ^ ⤸
```

Copying and Appending to Files

In addition to occasionally reorganizing your files, you'll often want to combine them or edit an old file. The COPY command allows you to add text to a file, build one large file from smaller ones, and duplicate a file within the same directory.

Let's use the COPY command to augment MACRO.CLI, currently in the user directory, :UDD:ALEXIS. First create another command file (macro) called MACRO2.CLI. Later we'll append MACRO.CLI to it. Type

```
) cre/i macro2.cli ↵
)) write The time is [|time]. ↵
)) write Today is [|date]. ↵
)) write My working directory is [|directory]. ↵
)) ) ↵
```

Now try it; type

```
) macro2 ↵

The time is 14:40:45.
Today is 03-SEP-85.
My working directory is :UDD:ALEXIS.
```

The new macro called MACRO2.CLI contains several bracketed commands called pseudomacros. Basically, pseudomacros act as variables for which the CLI will supply the actual information when it executes the macro.

The next step is to combine MACRO.CLI with MACRO2.CLI. The COPY command with the /APPEND switch directs the CLI to add to an existing file. To combine the files, enter the command, the destination file — the one to which you'll append text, and then the source file. Type

```
) copy/a  macro2.cli  macro.cli ↵
```

Now try the new macro. Type

```
) macro2 ↵

The time is 14:45:54.
Today is 03-SEP-85.
My working directory is :UDD:ALEXIS.
VS.PRACTICE contains:
Greetings from the AOS/VS session.
```

Pretty good! Now you have a macro that displays the current time, date, your working directory's name, and greetings from VS.PRACTICE in one command.

The COPY command is useful because it copies the *contents* of a file to the destination file, without the name, date and or other statistics of the source file. MACRO2.CLI contains MACRO.CLI but there's no trace of MACRO.CLI's name or creation date.

Deleting Files

Whenever you're finished with files or directories, it's a good idea to copy them to a backup medium, such as magnetic tape or diskettes, and then erase them from disk. (We explain backup procedures later in this session.) Disk space is valuable, and can easily be used up.

You'll use the DELETE command to erase files and directories. For example, we currently have an empty directory called RECORDS. To delete RECORDS, type

```
) delete/v records  ↵
DELETED records
```

Many times you'll gingerly delete files, making sure any needed files are left intact. The /C switch on the DELETE command directs the system to wait for confirmation before it deletes a file. For example, in :UDD:ALEXIS there are two macros: MACRO.CLI and MACRO2.CLI. But since MACRO.CLI is a subset of MACRO2.CLI, you don't need it anymore. Cautiously delete MACRO.CLI, by typing

```
) delete/v/c  macro.cli  ↵
macro.cli ?
```

Confirm the deletion. Type

```
y  ↵

DELETED  macro.cli
```

You can verify that only one macro remains with the FILESTATUS command. Type

```
) f/as/s  +.cli  ↵

DIRECTORY  :UDD:ALEXIS

  MACRO2.CLI   TXT    25-JUN-85    17:30:12   138
```

The FILESTATUS command displays only one file ending in .CLI.

Renaming Files

As you work on the system, you'll develop different versions of programs and documents and need to distinguish one version from another. The RENAME command allows you to modify a filename to identify its contents properly.

For example, you've learned a good deal about AOS/VS by now, so it's appropriate to rename the LEARNING directory to something more accurate, such as SESSION. To rename the directory, provide the name of the existing file, then supply the new filename. Type

```
) rename  learning  session ↵
```

Confirm the new directory name with the FILESTATUS command:

```
) f/as/s )

DIRECTORY :UDD:ALEXIS

MACRO2.CLI      TXT     ...
SESSION         DIR     ...
SESSION.LOG     TXT     ...
TESTFILE        TXT     ...
VS.PRACTICE     TXT     ...
```

Directory SESSION replaces directory LEARNING, leaving no trace of the old directory name.

Summary

We've covered a lot of ground in this section: creating, adding to, renaming, and deleting directories. At this point, if you completed the section sequentially, your directory structure should look like that in Figure 2-5.

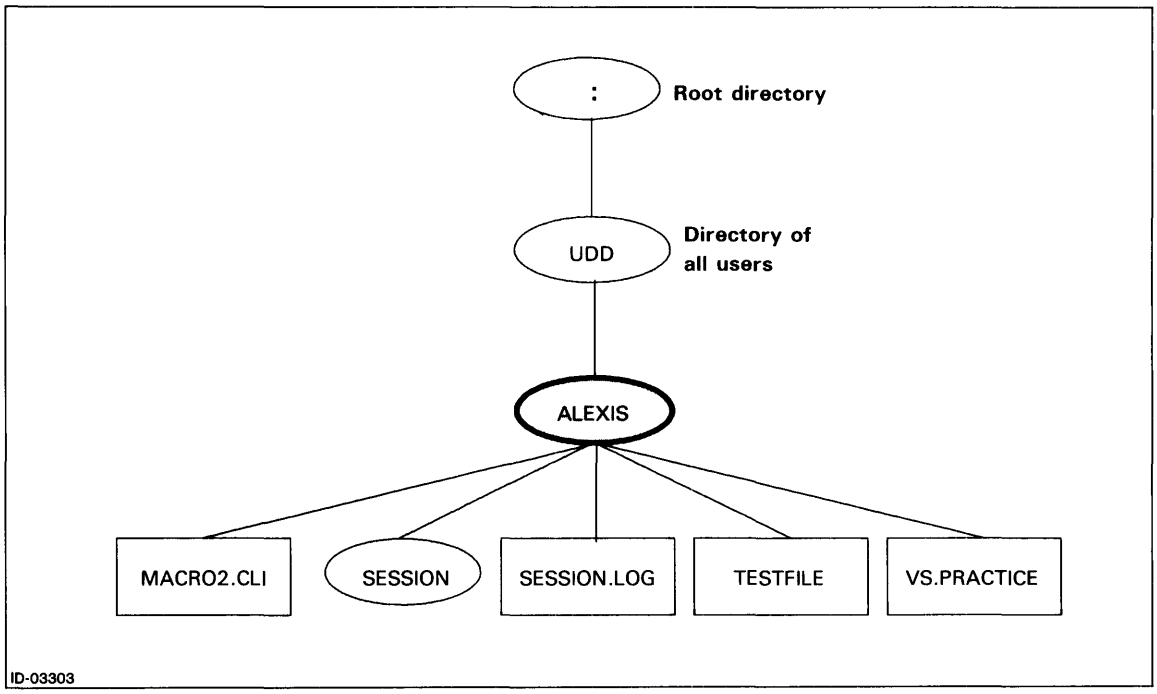


Figure 2-5. File Structure of :UDD:ALEXIS

Printing Files

All your work with AOS/VS doesn't have to remain on disk. Sometimes you might want paper copies of documents and programs — to mark up, store, or distribute. You will use the QPRINT command to print a file.

As the name implies, the QPRINT command sends a file to the printer; the prefix Q represents the term QUEUE. Because a line printer is a rather slow, shared resource, it can't always service your request immediately. So the system puts your request for a job on a waiting list, or a queue, for the device. As soon as your job comes to the head of the line, the printer services your request.

The printer prefaces each job with a header page, identifying the username, pathname, and time. The printed text of the file(s) follows.

To print one of our files, use the QPRINT command and include the filename as an argument. Type

```
) qpr macro2.cli ↓
QUEUED, SEQ=4409, QPR=127
```

The system confirms the service request, and displays the sequence number and priority of the job in the queue.

To see your position in the queue, use the QDISPLAY command. Type

```
) qd ↓
```

The system displays the waiting lists for different printers. The line printer is called LPT.

```
BATCH_INPUT      PRINT   OPEN

BATCH_OUTPUT     PRINT   OPEN

BATCH_LIST       PRINT   OPEN

LPT
*  4409     ALEXIS     :UDD:ALEXIS:MACRO2.CLI
   4410     CHRIS      :UDD:CHRIS:MORTGAGE.BASIC
   4411     FRAN       :UDD:FRAN:MORTGAGE.PAS

LQP              PRINT   OPEN
...
```

Another way to send system output to the line printer is by attaching the /L=@LPT switch to another command. The switch is handy for things like file sorts and program listings from compilers. Try it with the FILESTATUS command; type

```
) files/as/s/l=@lpt ↓
```

The system won't display a sequence number in the print queue, but you can use the QDISPLAY command to view it. Type

```
) qd ↓
```

And you'll see

```
...
LPT
* 4417    D    ALEXIS     :QUEUE:ALEXIS.0001.LPT

LQP              PRINT   OPEN

FTQ              FTA     OPEN

QWE              PRINT   OPEN

FLAGS EXPLANATION:
* = ACTIVE
```

The job, called 0001.LPT is in the queue for the line printer, LPT. The printer will handle your requests, so go to the printer and pick up your listing of files.

If your printer is frequently used, attach the /NOTIFY switch to the QPRINT command. This way, the system lets you know when your job is ready.

Handling the Line Printer

Each computer site has its own method of handling devices such as line printers or tape units. Sometimes you are allowed to work the device yourself. This is especially true if you have a small printer that sits on a desk. Other times an operator is in charge. Check with your system manager or operator (if there is one), and find out who is supposed to work the printer.

If an operator is in charge of the printer, he or she will tear the printed copies of your files off the printer, and hand them to you.

Without an operator, you need to work the printer yourself. For small line printers, it's a straightforward procedure. The printer resembles an electric typewriter, and to the right of the roll is a series of control buttons. To advance the paper and tear off your file, press the ON LINE button (taking it off line) and then press the FF button enough times that the file clears the printer. Once you tear off your file, press the ON LINE button again, so the printer is available for the next user.

On large, floor model line printers, it isn't as easy to see your printed file. So the challenge in handling a large printer is to maintain a correct paper fold — to keep the first page of a file on the outside, not the inside of the fold. When the printer finishes a job, it just waits for the next one; it doesn't advance the paper so you can tear off your job. You must advance the paper yourself, either manually or with a simple macro.

The best method is to use a form feed macro so that no one needs to work the controls. Ask your system manager if there is a macro. If so, return to your terminal and type its name (frequently it's FF); then go back to the printer and get your file. Use the macro in the future after you print a file.

If there is no form feed macro and the operator won't or can't do it, you must work the controls yourself. Press the ON/OFF LINE button to take the printer off line. Then press the TOP OF FORM button twice or four times — be sure to press the TOP OF FORM button an even number of times. This will ensure that the paper fold alignment remains as you found it. Then press the ON/OFF LINE button to put the printer back on line so it can continue serving users. Tear off your printed file.

Summary

Learning to use a line printer is pretty basic, and it's worth your time to become familiar with one. If you need assistance with other features of the device, see your system manager or refer to the user's guide accompanying the printer.

Protecting Files

AOS/VS has certain file protection features that allow you to define directories, or the files within directories, as public or private. Other AOS/VS protection features guard against unintentional file deletion.

Under AOS/VS you can choose to work in your own directories in complete privacy (even though there may be many people using the system), or, you can choose to share some directories or files with other users. You have the freedom to define a file as public or private.

This section explains the Access Control Lists (ACLs) that regulate who can use a file or directory. It also introduces the concept of permanence: a method of protecting a file from accidental deletion. Finally, the section describes file backup: a critically important procedure for periodically copying electronic files to magnetic tape.

Controlling File Access

File ACLs act as gatekeepers to a file. A file's ACL establishes who in the system can use it: who can execute it (if it's a program file), read it, and modify or delete it. AOS/VS assigns an ACL to a file when it's created.

The letters O, W, A, R, E represent the file privileges that are in effect. The privileges have the following meanings:

Privilege	Allows
O (Owner)	You can change the ACL (giving yourself or others any privilege), or delete the file or directory.
W (Write)	You can create and delete files, and change the ACL of files or the directory itself. With Write access to a file, you can modify the file's contents.
A (Append)	You can add files to a directory. With Append access to a file, you can get file status, a full pathname, and check permanence.
R (Read)	You can examine a list of files in a directory (FILESTATUS). You can also read the contents of a file (TYPE), providing you have Execute access to enter the directory.
E (Execute)	You can use the directory name in a pathname, and thereby reach a directory to which you do have O, W, A, R, or E access. Execute access to a file, such as a system utility, lets you execute it. The E privilege alone does not allow you to read filenames in a directory or read the text of a file.

Let's look at the ACLs for the session's files. Return to your initial directory, if necessary, and request a listing of ACLs for the files. Use the + template to indicate that you want a listing of all the directory files. Type

```
) dir :udd:Alexis ↵

) acl/v + ↵

=SESSION.LOG     ALEXIS,OWARE
=SESSION         ALEXIS,OWARE
=MACRO2.CLI      ALEXIS,OWARE
=VS.PRACTICE     ALEXIS,OWARE
=TESTFILE        ALEXIS,OWARE
```

All the files in :UDD:ALEXIS have the same ACL: OWARE.

Since the files from your AOS/VS session have the same ACL — username,OWARE — you have sole ownership and use of the files.

Since you didn't assign an ACL of username,OWARE to your files, who did? The system manager established a default ACL when he or she set up your account on the system. AOS/VS assigns this default ACL to all new files, unless you change the default. To verify the default ACL for our files, type the DEFACL command:

```
) defacl ↵
ALEXIS,OWARE
```

To change the default, you would use the DEFACL command with a list of authorized users and their privileges as arguments to the command. (Chapter 3 shows how.) To see how to assign user privileges, read on.

The owner of a file defines who can gain access to it for any reason. Suppose, for example, that Alexis wants to share a directory with other project members, and set up a common file within the directory where everyone can record project milestones.

In order to allow others to read or write to a common file, Alexis needs to give project members, in this case Chris and Fran, the privilege of using the directory :UDD:ALEXIS in a pathname. Execute access will allow them to pass through :UDD:ALEXIS and reach the subordinate directory, TRANSFER.

```
) acl  :udd:Alexis  Alexis,oware  Chris,e  Fran,e ↵

) acl/v :udd:Alexis  ↵

:udd:Alexis          ALEXIS,OWARE CHRIS,E FRAN,E
```

Now Chris and Fran can use :UDD:ALEXIS in a pathname, as shown in Figure 2-6, but they can't do anything with the contents of :UDD:ALEXIS. Next we create a directory subordinate to :UDD:ALEXIS for all to share. We'll call it TRANSFER, and give Chris and Fran more privileges. Type

```
) create/dir transfer ↵

) acl  transfer  Alexis,oware  Chris,are  Fran,are ↵

) acl/v transfer ↵

transfer             ALEXIS,OWARE CHRIS,ARE FRAN,ARE
```

With the Append privilege Chris and Fran can add files to the TRANSFER directory. The Read privilege entitles them to a file listing, using the FILESTATUS command. And the Execute privilege allows them to use the directory name TRANSFER in a pathname, such as :UDD:ALEXIS:TRANSFER:PROJECT.STATUS.

Now to create a group file:

```
) dir transfer ↵

) cre project.status  ↵

) acl  project.status  Alexis,oware  Chris,wr  Fran,w  ↵

) acl/v project.status ↵
project.status          ALEXIS,OWARE CHRIS,WR FRAN,WR
```

Alexis is the sole owner of the file PROJECT.STATUS; no one else can delete it or change its ACL. (And no one but Alexis has deletion privileges to the directory TRANSFER.) But now, other group members have Write access, allowing them to add to the file with the COPY command. Read access entitles Fran and Chris to display the file with the TYPE command. Execute access is necessary for program files, not for text files, so it's not assigned.

(Note that Write access does not allow Chris and Fran to use a text editor, because a text editor deletes the older copies of a file when it updates a file, and thus requires Write access to the directory. For this reason we suggest you use the COPY command with the /APPEND switch. Fran and Chris can move files into the TRANSFER directory, use the COPY/A command to append their file to PROJECT.STATUS.)

An alternative would be to assign Fran and Chris OWARE privileges to the file PROJECT.STATUS, so they could move it into their own directories to modify it.

The project group — Alexis, Fran, and Chris — can now keep files in a common directory, TRANSFER, and keep each other up-to-date on current objectives and milestones. Alexis's other files and directories continue to be private, safe from other's eyes.

The system manager will set ACLs for system files and directories. By experimenting with the DIRECTORY and TYPE commands, you can quickly discover which areas of the system are open to the public and which are private. For example, to obtain a file listing of the utilities directory :UTIL type

```
) f/as/s :util:#   ↵
WARNING: FILE ACCESS DENIED, FILE =
```

Or, if your system is more open, you'll see a listing of the files in directory UTIL.

Sometimes the *FILE ACCESS DENIED* message means the ACL is outdated. For example, if someone sends you a file that is set to their ACL, you'll need to adjust it before you can work with the file. Assume that you just loaded a file called $MAY.RATES from tape, and tried to move it from :UDD:ALEXIS into your directory SESSION. The following sequence is likely to occur:

```
) move/v  session  $may.rates  ↵
WARNING: FILE ACCESS DENIED, FILE=$MAY.RATES
```

To discover the ACL, type

```
) acl/v $may.rates   ↵
$may.rates        CHRIS,OWARE
```

Since the file is now in your directory, you have license to change it. Type

```
) acl/v  $may.rates  Alexis,oware') ↓
$may.rates
```

Now you, not Chris has sole access to the file. If Chris sent a number of files, you could change file ACLs globally by typing

```
) acl/d  +  ↓
```

The /D (default) switch on the ACL command assigns the current default ACL to all files in your directory (+). As you remember, the default ACL is set with the DEFACL command. In this case, Alexis's default ACL is ALEXIS,OWARE, so all files in the directory now have this ACL.

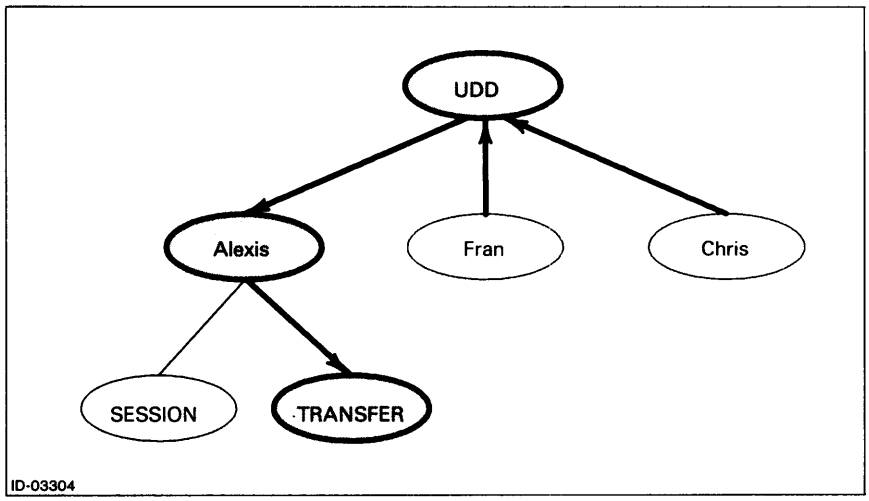


Figure 2-6. Path to a Public Directory

Making Files Permanent

You can further protect your own files with the PERMANENCE command. Whenever you stipulate that a file is permanent, the system will not delete it until you revoke the command (or someone with the privilege to do so deletes the superior directory). The PERMANENCE command ensures your files against careless use of a template or some other accidental deletion.

However, it's necessary to use the PERMANENCE command with discretion because a permanent file cannot be modified. Text editors and other utilities internally delete files and recreate them, so permanence can interfere with your work. You may want to set permanence only on directories and selected files. Note that permanence will not save a file if its parent directory is not permanent and you delete that directory.

Let's experiment with the PERMANENCE command. Type

```
) cre/dir  Pascal  ↓
) perm  Pascal  on  ↓
) perm  Pascal  ↓
Pascal  ON
) del/v  Pascal ↓
WARNING: CANNOT DELETE PERMANENT FILE, FILE Pascal

) perm  Pascal off ↓
) del/v  Pascal ↓
DELETED Pascal
```

The PERMANENCE command accepts templates, so you can set it easily for groups of files. For example, to protect all source files, you might type

```
)  perm  +.f77  on  ↓
```

Copying Files To Tape

At some point, you'll want to know how to copy your files to tape for safekeeping — a practice called file "backup." (Computers are very sensitive machines, and if by chance, some part malfunctions, you could lose valuable data; for this reason, each site should regularly copy files to tape.)

Most often, one person — such as a system manager or operator — is responsible for doing scheduled backups of the complete system. If this is the case, all you need to copy are your special files, and on occasion, do a backup of your directories (especially if it's bad weather).

It's good practice to keep a tape copy of all important information. We also recommend that you keep any sensitive information on tape, rather than leaving it on-line. (There are always a few privileged users, such as system managers, on the system who could read it.) Whenever you need to modify the file, you can load it into your directory. Be sure to keep tapes in a safe place: as you've seen, anyone loading the files from tape into their own directory can change the ACLs and read the files.

If you don't want to do tape backups now, you can skip this section and go on to "Obtaining Help on the CLI." To learn about tape backups, read on.

Your system probably has a procedure for backing up user files, and the backup medium is likely to be magnetic tape. (But it could also be diskettes or another disk.) Tape units are useful and valuable devices, but users on your system may not be allowed to operate them. (The amount of access users have to devices depends on your system: the operator may run all devices, or run tape units only; or there may be no operator, in which case you will be operating both the line printer and the tape unit yourself.)

The usual way to have a tape mounted on the unit is with the MOUNT command; it sends a mount request to the operator's console. Type

```
) mount tape   Could you mount a tape? ↓
```

Now, one of four things will happen:

- You receive the message *ERROR: OPERATOR NOT AVAILABLE*. If this occurs, skip to the section below: "If an Operator Doesn't Respond."
- You see the message *ERROR: REQUEST REFUSED...*. This may be followed by an explanation. If the explanation doesn't clarify things for you, go to the operator and ask about procedures. Having talked with the operator, you may want to try again later, or skip all the way to "Obtaining Help from the CLI."
- After 30 to 60 seconds, you receive the CLI prompt) in response to the MOUNT command. This means that an operator is on duty, has noted your request, and has mounted a tape for you. The name of the tape unit is a *link name*, determined by your first argument to the MOUNT command. In this case, because you typed MOUNT TAPE, the tape link name is TAPE. Go to the next section, "If an Operator Responds."
- No response. If, after a minute or so, the CLI prompt has not returned, you can assume that an operator is not on duty. (An operator can issue the CX OPERATOR OFF command to inform the system he/she is not on duty. If this command had been issued, you'd have received the message *OPERATOR NOT AVAILABLE*. Since the command was *not* issued, you might wait indefinitely for a response to a MOUNT command.) Type CTRL-C CTRL-A to interrupt the MOUNT command, and then type DISMOUNT TAPE and skip to the section below: "If an Operator Doesn't Respond."

If an Operator Responds

If the CLI prompt) returns in response to your MOUNT command, type

```
) dump/l=@lpt  tape:0  ↵
```

The DUMP command copies (dumps) disk files to a destination file. If you type the DUMP command from your user directory and omit filenames, the CLI copies *all* files from the user directory and subordinate directories. In this case, the destination file is tape file 0 (its link name can be TAPE file 0 or TAPE:0). The /L=@LPT switch requests that a listing of all copied files be made by the line printer.

When the system has dumped all your files to tape, the prompt will return. To ask the operator to dismount the tape, type

```
) dismount tape  thanks  ↵
```

This command requests the operator to dismount the tape.

If an Operator Doesn't Respond

If you received an *OPERATOR NOT AVAILABLE* message or no message at all in response to the MOUNT command, you'll need to mount the tape and operate the unit yourself.

There are two kinds of tapes: cartridge and reel-to-reel. A cartridge tape is rectangular, about 4 by 6 inches, and fits into a slot in the front of the tape unit. Usually, a cartridge tape unit is fairly small, fitting on or under a desk. A reel tape is round, of different diameters, and fits onto a round holder in the tape unit. A reel tape unit is a cabinet — roughly the size of a refrigerator.

If you have a cartridge tape unit, follow these steps:

1. Locate a cartridge tape, and open the write-enable dial, labeled SAFE, that's in the upper right corner. This allows the system to write to the tape.
2. Insert the cartridge into the tape unit slot: hold the cartridge so the clear plastic side is on your right. Firmly push the tape into the slot.
3. You'll see the READY light go on and hear the tape advance. Once advanced, copy your files with the DUMP command line, as shown below. Ask your system manager for the tape unit name — it's either MTC or MTJ.

If you have a reel-to-reel tape unit, follow these steps:

1. Locate a reel of magnetic tape that has a yellow or black plastic ring in it. This is a "write-enable" ring that permits the system to write on the tape.
2. Bring the tape to the system's tape unit. If there are several units, find an empty one. Open the clear plastic door to the unit, if any.
3. If the unit capstan (the tape holder) is *below* the take-up reel, the unit is type MTB or MTD: a diagram on the unit door shows how to thread the tape. If the capstan is on the *right*, *alongside* the take-up reel, the unit is a "streaming" type MTC, and a diagram above the capstan shows how to thread the tape. Mount and thread the tape according to the diagram. In either case, rotate the take up reel 4 or 5 times to get a good grip on the tape.
4. On the central panel of the unit, there are three rocker switches. The POWER switch should be ON. Press the BOT/UNLOAD rocker switch to BOT. The tape will hesitate, then move forward and stop.
5. Press the ON LINE/RESET rocker switch to ON LINE.
6. If the unit is an MTB, note the number on the unit thumbwheel (the dial's set to 0, 1, or 2, etc.), because you'll insert this number in the filename (as in MTB0 or MTB1). Then return to your terminal. An MTC and an MTD has no thumbwheel; assume the unit is number 0, so the filename is MTC0 or MTD0.

Having physically mounted the tape, you can dump your files. You'll use the actual tape unit device name for this. Type

```
) dump/l=@lpt @mtc0:0 ↵        (Or @mtj0:0 or @mtb0:0 or @mtd0:0)
```

The @ specifies the peripheral directory, shown in Figure 1-3. The peripheral directory is the home of all device files — for instance, the line printer is @LPT, the tape unit, @MTB0.

The DUMP command copies (dumps) disk files to a destination file. If you type the DUMP command from your user directory and omit filenames, the CLI copies *all* files from the user directory and subordinate directories. In this case, the dump is to tape file 0, at @MTJ0:0, @MTC0:0, @MTBn:0, or @MTD0:0. The /L=@LPT switch requests a listing of all copied files be made by the line printer.

When the system has dumped all your files to tape, the prompt will return. To rewind the tape, type

```
) rewind @mtc0 ↵        (Or @mtj0 or @mtb0 or @mtd0)
```

Return to the tape unit to remove the tape. Pull a cartridge tape from the unit. With a reel tape, first press the RESET switch, then the UNLOAD switch, and then remove the tape.

After Dumping to Tape

You have dumped all your files to tape. You can retrieve the tape from the tape unit yourself, as in the previous section, or from the operator. Tear the file listing from the line printer. Record the *file number* holding the files on the paper listing, and clip the listing to the tape.

Having done all this, put the tape and listing in a safe place.

The tape file number (in this case 0) is very important. If you forget it and inadvertently dump to the same tape file number later on, you'll lose the original tape file and all subsequent tape files. For your next dump on this tape, copy to file 1, and then 2, and so on. If the tape is long enough, you can have as many as 99 files.

For later dumps, if you need to do them, you can use date switches to dump only those files that have changed since your last backup. For example, tomorrow you might type

```
) dump/l=@lpt/after/tlm=10-JAN-86  @mtc0:1  ↵
```

This command line dumps all files modified on or after 10 January 1986 to the *second* file on the tape. The date switch /TLM= means Time Last Modified. Instead of @MTBn:1, you may use LINKNAME:1 or @MTC0:1 as the destination file.

If you ever need to restore your dumped files to disk, you'll use the LOAD command. Chapter 3 describes it.

Having read this entire section, you may find that you never need to dump files to tape. This is fine as long as someone on the system does regular backups. But if you need to do it, you know where to find instructions, whether or not an operator is on duty.

Summary

AOS/VS is a secure system because only authorized people, with legitimate usernames/passwords, can log on. Once on the system, ACLs protect your private work from scrutiny and unauthorized changes and deletions, yet permit public use of programs, such as the CLI program, text editors, and project files.

With the PERMANENCE command AOS/VS allows you to protect your files from accidental or malicious deletion. And when files are very sensitive, you can copy them to magnetic tape (or diskettes) and keep them in a safe place.

Obtaining Help from the CLI

AOS/VS has an extensive Help facility. Whenever you're unsure about a command — its format, switches, or arguments — simply type the HELP command. We didn't describe the command earlier because you need a little background to understand its messages.

The HELP command can display a brief summary of a command's format and switches; for example, type

```
) help permanence  ↓

PERMANENCE      - REQUIRES ARGUMENT(S)
  SWITCHES: /1= /2= /L(=) /Q /V

FOR MORE HELP TYPE
HELP/V permanence
```

Adding the /V switch to the HELP command gains you a fuller (verbose) explanation. There's a HELPV macro that's faster to type. The detailed Help message goes beyond the summary, and explains the meaning of the command and its switches, concluding with examples. Because the Help messages often span several screens, use CTRL-S to freeze the display, CTRL-Q to advance it. Try it; type

```
) helpv permanence  ↓

                P E R M A N E N C E   C o m m a n d
Format:
PERMANENCE  pathname  [ON ]
                     [OFF]

Displays or sets permanence for one or more files.  When you turn
permanence on, it protects file(s) from accidental deletion,
unless you delete a parent directory whose permanence is off.  You
can use templates in the pathname argument.  If you want to delete
file(s) with permanence on, turn permanence off, then delete the
file(s)....
```

As you program under AOS/VS, you'll be able to use the Help facility for information on a wide range of topics. Help is available on different utilities and editors that run under AOS/VS, as well as on the CLI.

The HELP command without arguments, produces a complete listing of the kinds of information available. Type

```
) help )

TOPICS ARE:

*1_SWITCH       *2_SWITCH       *ACCESS         *AFTER_SWITCH   *ANALYZE
*BAS            *BUILD          *CALC           *CC             *CCL
*CCL_OPT        *CC_GLOBAL      *CC_LOCAL       *CHECKIN        *CHECKOUT
.
.
.
FOR MORE HELP ABOUT ANY ITEM ABOVE, TYPE 'HELP *ITEM'
```

From the complete list, you select a specific topic by including it as an argument to the command; for example, to learn about the 1_SWITCH, type

```
) help *1_SWITCH )

ALL CLI COMMANDS ACCEPT /1=XXX AS A COMMAND SWITCH.  YOU MUST SUPPLY ONE
OF THE FOLLOWING FOR XXX:

IGNORE
WARNING
ERROR
ABORT

THIS SWITCH WILL CAUSE THE HANDLING OF CLASS1 EXCEPTIONS TO BE SET TO XXX
FOR THE DURATION OF THE COMMAND ON WHICH IT APPEARS.
```

Summary

You'll find the on-line Help facility very useful in program development and your everyday use of the system. It comes in two levels: a summary of command format and a detailed description of format and use. You'll often use Help to obtain a listing of command switches.

Batch Processing

Throughout this session you've used AOS/VS interactively. That is, you've issued commands to the CLI from the terminal keyboard, and the CLI has displayed responses on your terminal screen.

However, AOS/VS also supports noninteractive processing, known as batch processing. With this kind of processing you create a file of commands, and submit it to the CLI. The CLI sets up a new process, called a *batch job*, executes the process, and notifies you when the process is complete.

The beauty of batch processing is that it frees your terminal so you can continue to work on the system while a program is executing. For example, you can compile or assemble a program in batch and use your terminal to create another source file. Many sites use batch processing for large jobs such as payrolls, which consume a lot of processor time; the job can run unattended, often at night, when the system demand is low.

To run a batch job, use the QBATCH command. For example, to get a printout of the files in your directory, use the FILESTATUS command in in batch. Type

```
) qbatch f/as/s  ↵

QUEUED, SEQ=4213, QPRI=127
```

The CLI sets up a new process that executes the FILESTATUS command and sends the output to the line printer.

Because it's such a short job, it might complete before you can type the QDISPLAY command. But try it, type

```
) qd  ↵
BATCH_INPUT   BATCH   OPEN
 *7173  D  ALEXIS   :UDD:ALEXIS:?75.CLI.002.JOB
.
.
.
```

In this case the batch job is in the queue, and the system will run it as soon as possible. (Sometimes this is a few minutes, other times batch jobs might only run after 6:00 pm.). The CLI sends the output of the job to the printer, and records any errors in a list file.

If the batch queue is open, your job is complete. Go to the printer and pick up the listing. If you look at the middle of the listing — after the header — you'll see the CLI printed and executed the FILESTATUS command:

```
) f/as/s

DIRECTORY  :UDD:ALEXIS

MACRO2.CLI      TXT ...
RECORD.CLI      TXT ...
SESSION         DIR ...
SESSION.LOG     TXT ...
TESTFILE        TXT ...
VS.PRACTICE     TXT ...
```

Note that the batch output file resembles your terminal screen at logon. This is because a batch command sets up a new process, thus the listing of the AOS/VS banner, system messages, and other information. At the end of the file, *END OF FILE* is printed, as well as a notice that there were no errors sent to the LIST FILE.

You can also write a macro that sets up a batch job. The macro would list a number of CLI commands to perform. You use an /M switch with the QBATCH command, indicating that the commands to execute are included in the macro file. You *must type a right parenthesis as the last line* to signal the end of the file. (Because the macro must end with a right parenthesis, you need to create the file with a text editor, which we explain in Chapter 4.)

A batch macro works the same way as the FILESTATUS example above. Again, the CLI sets up a batch process, submits the macro as the input file, executes the macro, and sends the results to an output file, which is usually the line printer.

Suppose we wrote a macro called BOOKKEEPING.CLI in the SED text editor. It might look like this:

```
qbatch/m
write The files in [!dir] are
files/as/s
)
```

To run the macro, type

```
) bookkeeping   ↓
```

And that's how batch works. We use another batch macro in the Pascal chapter, in "Printing the Full Schedule" that shows how to pass terminal input to a program running in batch.

Summary

Batch processing lets you complete several jobs at a time. By compiling programs and large sorts in batch, you can free your terminal for other work and run processor-intensive jobs during periods of low demand.

Running Programs from the CLI

If you work exclusively in the CLI, you don't need this section. But much — if not most — of your work will involve system utilities: text editors, compilers, linkers, debuggers, and other program development tools.

Using AOS/VS Utilities for Program Development

AOS/VS supports many system utilities, and the balance of this manual explains many of them. The chapters that follow explain the SED and SPEED text editors, the compilers for AOS/VS BASIC, FORTRAN 77, COBOL, Interactive COBOL, Pascal, and C, plus the BASIC and Business BASIC interpreters, the Macroassembler and the Sort/Merge utility.

Other utilities are available to you with AOS/VS. One utility called DISPLAY allows you to examine files with nonprinting characters, such as program files, which the TYPE command can't display; it will even display hexadecimal values in octal. Other file utilities are FILCOM, which compares files with nonprinting characters word by word — 16 bits at a time, and SCOM, which compares source files line by line and compiles a list of file differences.

There is one copy of each utility on the system, and it's usually kept in a directory called UTIL. Figure 2-7 shows the relationship of directory UTIL to user directories and shows the path from ALEXIS to UTIL.

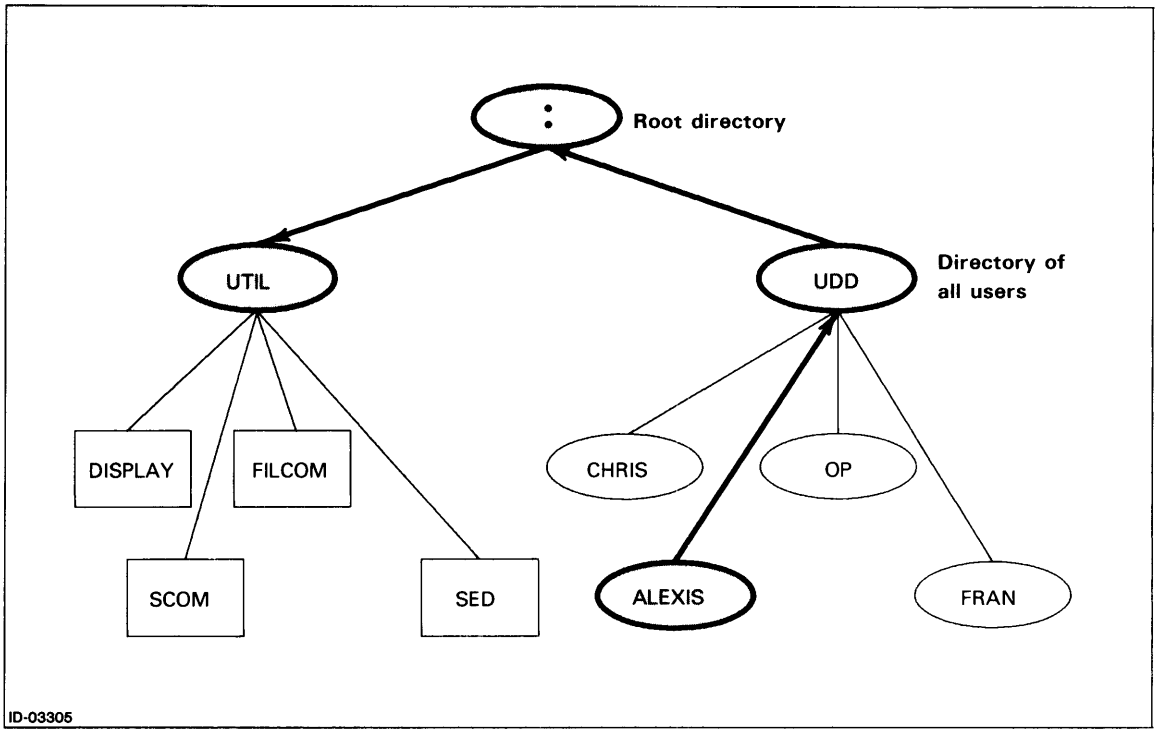


Figure 2-7. The Path to Directory :UTIL

Finding System Resources with Search Lists

Earlier, you experimented with pathnames and found that the system can locate a file in any directory if you provide a route (or a path) to the file. Typing long pathnames, however, becomes cumbersome, so each user has a search list. A search list is a short list of directories that the CLI scans if it can't find the specified file in the working directory. The default search list is set by a SEARCHLIST command within the log-on macro that the system manager provides. But you can tailor your searchlist in a log-on macro to whatever you want. Your searchlist is private, no one but you can display it.

To display your current search list, type

```
) searchlist ↓
:MACROS, :, :UTIL, :UTIL:CEO_DIR
```

There are four directories on Alexis's search list — MACROS, the root (:), UTIL, and the CEO directory. Your own search list may well include directories other than :UTIL, but most search lists include :UTIL.

Besides freeing you from typing long pathnames, search lists allow you to run different system utilities without knowing exactly where the file resides. For example, execute the SED text editor from the CLI. It's traditionally kept in :UTIL:

```
) xeq sed ↓
Name of file to edit:
```

Leave the text editor with the CTRL-C CTRL-B:

```
CTRL-C CTRL-B
```

... (Program abort message) ...

You executed SED without a full pathname. Now, see what happens when you remove UTIL from our search list. Reset the list to your SESSION directory:

```
) searchlist :udd:Alexis:session ↵

) xeq sed ↵
ERROR:  FILE DOES NOT EXIST
xeq,sed
```

You receive a *FILE DOES NOT EXIST* message because the system can't find the SED program. If you ever get this or some other obscure error messages when you try to execute a program that you think exists, check your search list. It may be wrong.

To correct the search list, type your original one. In this case, type

```
) sea :macros, :, :util, :util:ceo_dir ↵
```

You can use the pseudomacro !SEARCHLIST to save time in adding to a search list. For example, augment your list to include directory SESSION. This way, you won't have to use any long pathnames. Type

```
) sea session [!search] ↵

) sea ↵

:UDD:ALEXIS:SESSION,:MACROS,:,:UTIL,:UTIL:CEO_DIR
```

By adding :UDD:ALEXIS:SESSION to your search list, you can gain access to any file in SESSION without typing a full pathname.

Summary

Now you've executed the SED utility and experimented with search lists, which is another way to gain access to files without using long pathnames.

Security Check List

AOS/VS serves a community of users. As a community, all users are responsible for the security of the system data. In addition, all users must help protect their files, directories, and passwords.

Throughout the session, we've mentioned methods of protecting files and directories. Here we gather the suggestions together so you can check off which ones you and your community should follow. Of course all these practices are site-specific.

Because the number of users and the sensitivity of work varies widely from site to site, you'll find some are very controlled — users have access to little beyond their user directories and never touch the system hardware. Then, there'll be AOS/VS sites that have rather informal control: users will work all the hardware devices and move freely through many directories.

Depending on the type of site you have, you'll want to make some or all of the following practices second nature:

- Keep your password secret. Passwords are the most important line of defense. Regardless of the formality or informality of your system, never share your password with another user. If for some reason you feel your password is publicly known, change it immediately. Choose passwords that are obscure — avoid obvious ones such as your children's names or your initials or your hobbies. Every few months, it's a good idea to change your password, just to ensure privacy. The first section of the session, "Taking the First Steps" describes how to create and update passwords.
- If you're logged on, don't leave your terminal unattended for long periods of time. Anyone could tamper with your files or gain access to other directories in the system, where harm could occur.
- Keep highly sensitive material on tape (or diskette) and store it in a secure place. There are always privileged users on the system who can gain access to your files. The chapter's section on "Protecting Files" describes how to copy files to tape.
- Set the ACLs of files and directories carefully. In some cases, you won't want anyone to access a file; in others, you'll want to provide a path to a file. Refer to the "Protecting Files" section for guidelines on setting ACLs.
- Keep an accurate default ACL. Since this is the ACL assigned all new files and directories, update it as your work groups change. Chapter 3 describes how to set your default ACL.
- Protect critical files by making them permanent. This way they can't be carelessly or maliciously deleted.
- Be observant of what goes on around you. Notice the dates and times of previous logons or file modifications. Each time you log on, the system displays the time your last computing session began. If the report is erroneous, speak to your system manager. Someone might be using your account.
- If you're responsible for a directory, keep a sharp eye out for any changes. Notice the last time files were modified; and, if you see any peculiar activity, notify the system manager. Also set the directory ACL shrewdly — perhaps giving Owner (and therefore delete privileges) only to yourself. You must beware of a Trojan Horse: a perpetrator who gets into your directory and substitutes an .OB or a .PR file for one of yours or modifies one of your files. If such an intrusion occurs, when you run the program, it can perform maliciously: the program can read sensitive information and record it in another directory. If you suspect such behavior, set up a log file and monitor the program and speak to your system manager.

All these preventive measures will ensure that your AOS/VS system is secure from accidental or malicious intrusion.

What's Next?

Chapter 3 provides a more detailed description of each CLI command that the session introduced. Chapter 16 is a guide to documentation on AOS/VS products; it will show you where to turn for further information.

Next, you might turn to the text editing session in Chapter 4. Or begin exploring one of the language chapters, which show how to develop and run a program under AOS/VS.

End of Chapter

Chapter 3
Common CLI Commands

This chapter is primarily a reference chapter, providing information on the CLI commands introduced in Chapter 2's session with AOS/VS.

The first sections summarize

- CLI syntax
- Special CLI control keys and templates.

The balance of the chapter summarizes 28 CLI commands. It is meant as an introduction to the *Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS)*, which explains all CLI commands and their special features.

CLI Command Syntax

The AOS/VS session in Chapter 2 explains the CLI command syntax, and leads you through many exercises demonstrating it. The sections that follow summarize the basic rules.

As you saw in Chapter 2, the CLI is a powerful utility. It provides over 80 commands and a macro facility so you can create even more. You can do a lot of work using just a few of these commands.

Abbreviations

You can abbreviate a CLI command to its shortest unique string. For example, the FILESTATUS command can be entered as F, the CREATE command can be entered as CR. Don't be afraid to experiment with short abbreviations; the worst that can happen is an error message such as

```
ERROR: COMMAND ABBREVATION NOT UNIQUE
```

which means that you must type more letters to identify the command. Within this book, we don't always show the shortest abbreviation; instead, we tend to use the shortest abbreviations that have mnemonic value.

Command Switches

You can add switches to a command to modify its execution. All switches begin with a slash (/). For example, the /ASSORTMENT switch on the FILESTATUS command requests a list of statistics with the files. An /L=@LPT switch requests the CLI to send any output to the printer.

The Basics of Command Lines

A CLI command line follows the format:

command*[/switches]* required-arguments *[optional-arguments]...*

You always type a command next to the CLI) prompt, and enter it by pressing the NEW LINE key. (It's the CLI prompt that signals you when the system is ready to receive and execute your command.) The CLI is space sensitive: no space can exist between the command and its switches. However, a space, comma, or tab must separate arguments from the command. Extra spaces or even tabs within CLI commands have no effect. You must end all command lines by pressing either NEW LINE (⤶) or CR (Carriage Return).

Multiple Commands and Long Command Lines

You can put many commands on a line, as long as you separate each command with a semicolon. For example,

```
) files/as vs.practice; ty vs.practice  ↵
```

A command line can span several lines. To continue a command onto another line, type an ampersand (&) and press NEW LINE. On the next line, the CLI will display an ampersand prompt, and you can continue typing. When you want the CLI to execute the command string, press ↵ without typing an ampersand. For example:

```
) ty& ↵
& ) pe& ↵
& ) vs.practice  ↵
```

Getting Help About CLI Commands

The CLI has an extensive Help facility. You can obtain a display of all CLI commands with the command HELP *COMMANDS. For details on any individual command, type HELPV command-name.

CLI Errors and Error Messages

When the CLI cannot execute a command, it displays an explanatory message called an *error message*. Then it stops and waits for another command. Generally, CLI errors do no harm; everyone makes them, and usually all you need to do next is retype the command correctly.

CLI error messages begin with the word WARNING, ERROR, or ABORT. The text that follows these words describes the error; for example,

```
WARNING: FILE DOES NOT EXIST, FILE filename
```

If the explanatory text allows you to understand what went wrong, you can correct it and continue with what you're doing. If you *cannot* understand what went wrong, look up the error text in the *Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS)*.

CLI Control (CTRL) Keys and Templates

The CTRL key, in conjunction with another key, notifies the CLI to take special action. Some key combinations will reposition the cursor, allowing you to modify text; other combinations alter the display of system output on the screen. CLI template characters represent different character strings; they're most often used in arguments to a CLI command.

Table 3-1 describes the keys used for simple editing in the CLI; Table 3-2 describes the key sequences for controlling system output. (In each case, you enter the keys the same way: press down the CTRL key, and while depressing it, type the second key.)

Table 3-1. Screen Editing Keys

Sequence	Effect
CTRL-A	Redisplays the last command typed.
CTRL-B	Moves the cursor backward to the last letter of the preceding word.
CTRL-F	Moves the cursor forward to the first letter of the next word, or redisplays the next word of the last command typed.
→ key or CTRL-X	Moves the cursor forward one character.
← key or CTRL-Y	Moves the cursor backward one character.
CTRL-E	Opens a line for new text, or closes an insert.
CR or ERASE EOL keys	Deletes all characters to the right of the cursor. CR executes the command.
DEL key	Deletes the preceding character.

Table 3-2. System Control Sequences

Sequence	Effect
CTRL-L	Clears the screen, and directs the CLI to execute the command (if any).
CTRL-S	Freezes a display. To restart suspended information, enter CTRL-Q.
CTRL-Q	Resumes a display suspended by CTRL-S.
CTRL-O	Discards the output, so there is no display. To resume display, enter CTRL-O again.
CTRL-U	Erases the current CLI command line.
CTRL-C CTRL-A	Interrupts execution of the current command.
CTRL-C CTRL-B	Stops the current program, and returns control to its parent process. If entered from the CLI, it logs you off the system.

Table 3-3. CLI Template Characters

Template Character	What It Does
* (asterisk)	Matches any single character except a period.
- (hyphen)	Matches any character string that does not contain a period.
+ (plus sign)	Matches any charcter string, including strings with a period and null strings.
# (number sign)	Expands to + and +:+ and +:+:+ and so on, until all subordinate directories have been matched.
\ (backslash)	Tells the CLI to exclude any filenames that match the following template.

Dictionary of Common CLI Commands

Table 3-4 lists the CLI commands in this dictionary.

Each command in the dictionary is followed by its meaning (what it does) and its format (how you write it), the reasons you might use it, and an example. The more commonly used switches are listed with each command. For a full list of switches, refer to each command in the *Command Line Interpreter (CLI) User's Manual.*

Table 3-4. Common CLI Commands

Command Name	What It Does
ACL	Controls access to files and directories.
BYE	Logs you off the system, or stops the current program.
COPY	Duplicates a file, or combines several files.
CREATE	Makes a new directory or file.
DATE	Displays the system date.
DEFACL	Sets or reports a user's default ACL.
DELETE	Eliminates one or more files.
DIRECTORY	Displays or changes the current working directory.
DUMP	Copies one or more files to a destination file.
FILESTATUS	Lists file and directory names and related statistics.
HELP	Provides information about CLI topics and commands.
LISTFILE	Sets or displays the current list file.
LOAD	Restores one or more files, copied with the DUMP command, to disk.
LOGFILE	Sets or displays the current log file.
MOVE	Copies one or more files to a different directory.
PERMANENCE	Displays current status of Permanence; makes a file permanent when turned ON, or returns it to an impermanent state when turned OFF.
QBATCH	Creates and submits a batch job.
QDISPLAY	Shows where jobs are in batch and print queues.
QPRINT	Puts one or more files in the print queue.
RENAME	Gives a new name to a file.
SEARCHLIST	Sets or displays the directories the CLI will scan for a file.
SPACE	Displays the amount of used and unused disk space.
TIME	Exhibits the system time.
TYPE	Displays one or more files on the terminal screen.
WHO	Describes a process.
WRITE	Sends text to the screen or a file.
XEQ	Executes a program.

ACL

Sets or displays the Access Control List (ACL) for a file.

Format

ACL*[/switches]* filename *[username,access-privileges]* ...

The ACL command with only **filename** as an argument displays the file's ACL. Each directory and each nondirectory file has an ACL, specifying who has access to the file and what type of access they have. With *username,access-privileges* arguments, the command sets a new control list for the file. These arguments are paired: *username,access-privileges*; and you can use templates with the username portion of the argument.

The types of access-privileges are as follows.

Access Type	File Type	Allows	Proviso
O (Owner)	Directory	You can change the ACL, rename or delete the directory, regardless of the ACLs of files within it. This permits you to change the ACL of any file (directory or nondirectory) within the directory.	You need Execute access to enter the directory and use its name in a pathname.
	File	You can change the ACL, rename or delete the file, check file status, check or change Permanence, create, read, edit or delete the user data area (UDA) for the file.	You need Execute access to the directory.
W (Write)	Directory	You can delete or rename any file *within* the directory, regardless of its ACL. You can create files and change file ACLs.	You need Execute access to the directory.
	File	You can change file contents, get a complete pathname, check file status, check or change Permanence, create or edit a user data area (UDA).	You need Read access to see what's in the file, and Write access to the parent directory, if you're using a text editor.
A (Append)	Directory	You can add directories or files to it.	You need Execute access to the directory.
	File	You can check file status, get a complete pathname, and check or change Permanence for the file.	None.
R (Read)	Directory	You can read a file listing, or read the ACLs of files *in* the directory.	You need Execute access to the directory.
	File	You can read the file (with the TYPE command or a text editor), check the file status, get a complete pathname, and check or change Permanence. You can also copy a file or debug a program.	You need Execute access to the parent directory and all higher directories.

(continues)

Access Type	File Type	Allows	Proviso
E (Execute)	Directory	You can enter the directory and use its name in a pathname; you can also list (FILESTATUS) a specific filename if you already know it.	You need Read access to list files and ACLs.
	File	You can execute a program file with the XEQ or PROCESS command, read file status, get a complete pathname, and check or change Permanence.	You need Read access to debug or edit a program file.
„ (null)	Directory	You can't gain access to files within the directory. By default, users have null access to other users' directories. However, if you grant access to a group of users with a template, null allows you to exclude individuals within that group.	
	File	You can't gain access to the file. Used as explained above.	

(concluded)

By default, you have OWARE access to your own files. The main reason to change an ACL is to give other users access to your files. When you change an ACL of your files, give yourself OWARE access privileges to your files; and if others need to use a file, give them Execute privileges to the directory and all superior directories, and Read access to the file itself.

Username can be literal, such as Alexis, or it can include template characters +, −, or * to specify a group of users. For example, :UDD:$+ includes all users whose usernames start with the character $. When you give access privileges to different users, be sure to assign specific username privileges before general (template) assignments. For example, assign CHRIS,OWARE before +,RE. By default, users who are not given access have null access.

However, if you use a template to give a group of users access, Null lets you *exclude* individual members of the group. You must assign Null access to specific individuals before you give the group privileges. For example, you would assign all privileges to Alexis (ALEXIS,OWARE), then exclude user Fran ($FRAN,,), and finally assign Read and Execute to all users (but Fran) whose name begins with $.

Why Use It?

In a multiuser system, you need access to, and privacy for, your own files. You also need protection from accidental (or malicious) deletion. *Access control lists* (ACLs) help ensure the privacy and safety of files. They prevent unauthorized users from reading the files, or modifying them in any way. And of course, they ensure against accidental or malicious deletion.

With ACLs, you can selectively share files with other people on the system, without risking unauthorized file deletion or modification. ACLs also allow all users to run system utilities, without any risk or harm.

The default ACL for all files in your user directory is USERNAME,OWARE. You can change this for any CLI session with the DEFACL command. Many systems have a logon macro that's executed for each user when he or she logs on. If this macro is present in your initial user directory, you can use a text editor to insert a DEFACL command in this macro.

Files that are copied to tape or to your directory with a DUMP or MOVE command usually retain their original ACLs. So, if you receive files from another person, use the ACL command to give yourself OWARE privileges to them before trying to edit or otherwise process them. If you forget, you'll get an ***ACCESS DENIED*** message when you attempt to access the file.

Switches

/D Assign the default ACL (usually USERNAME,OWARE) to the file. The default is set with the DEFACL command.

/V Displays the file's pathname along with its ACL.

Examples

```
) acl  myprog.pr ↓
ALEXIS,OWARE
) acl  myprog.pr  Alexis,oware  Chris,, +,re  ↓
) acl myprog.pr  ↓
MYPROG.PR  ALEXIS,OWARE CHRIS, +,RE
```

The first ACL command line displays the ACL for the file MYPROG.PR. The second command adds Read and Execute access to the file for all users except Chris, who is assigned Null access (,). Note that the command assigns Null access to Chris *before* the general (+) assignment of Read and Execute access. The third command confirms the new ACL.

```
) acl :udd:Alexis ↓
ALEXIS,OWARE
) acl :udd:Alexis Alexis,oware +,e ↓
) acl :udd:Alexis  ↓
ALEXIS,OWARE  +,E
```

The preceding commands display and modify access privileges to directory :UDD:ALEXIS, giving all users Execute access to Alexis's directory. Now they can enter :UDD:ALEXIS and/or use MYPROG.PR in a pathname.

```
) defacl  ↓
ALEXIS,OWARE
) acl/d + ↓
```

The preceding commands display Alexis' default ACL (ALEXIS,OWARE), then restore the default ACL to MYPROG.PR — preventing any other user (except a user with Superuser privileges) from accessing MYPROG.PR.

BYE

Stops the current CLI process.

Format

BYE

The BYE command, when issued from the CLI, terminates the user process that the system created when you logged on. This logs you off the system. Issued from BASIC or a utility like SED or SWAT, BYE returns you to the CLI.

Why Use It?

Whenever you want to end a computing session, type the BYE command to log off the system. You'll also use the command to exit from many AOS/VS utilities such as the SED text editor and the SWAT debugger.

Switches

Switches are program-specific.

Examples

```
) bye ↓

AOS/VS CLI  TERMINATING     18-JUL-85      14:40:58

Process 0057 terminated.
Connect time 0:47:38
User 'ALEXIS' logged off @CON16   18-JUL-85   14:40:58
```

The preceding example shows Alexis logging off the system, while the following example shows a user exiting from the SED text editor.

```
*  bye ↓                                          page 1 line 345
------------------------------------------------------------------
Do you want to save the original file as a backup file?    n ↓
Output file - :UDD:ALEXIS:TESTFILE

)
```

COPY

Duplicates or combines one or more files.

Format

COPY *[/switches]* new-pathname old-pathname *[old-pathname]* ...

The COPY command duplicates the *contents* of a file, without the filename or creation date. It copies one or many file(s), named in **old-pathname**, into **new-pathname**. If you specify two or more old-pathnames, they will be copied in the order given. Unless you use the /APPEND switch, the COPY command first creates the new file, then writes to it.

Template characters don't work with the command; you must specify the entire name.

Why Use It?

The COPY command allows you to add text to a file or build one large file from smaller ones. With it, you can duplicate the contents of a diskette or a tape file onto a disk file, or vice versa. The command will also duplicate a file within a directory. Because it copies the *contents* of a file, without the name, creation date, and so on, it's preferable to the MOVE command.

Switches

/A Appends or adds **old-pathname** to the end of an existing file, named in **new-pathname**.

/D Deletes the file named in **new-pathname**, and create a new one with the same name; then it copies **old-pathname** to **new-pathname**.

/V Verifies or displays the names of the source files copied. This is useful when you copy many files into one large file.

Examples

```
) copy project_status March_report ↵
```

This command creates PROJECT_STATUS in the working directory, then it copies MARCH_REPORT to it. You can verify that the file exists with the FILESTATUS command.

```
) copy/a/v project_status ^June.budget ^July.budget ↵
^June.budget
^July.budget
```

This copies JUNE.BUDGET and JULY.BUDGET — which reside in a superior directory — to the existing file, PROJECT_STATUS in the working directory. The system verifies the source files copied.

```
) copy/v @dpj10 dfile ↵
dfile
```

This copies the file DFILE onto the diskette located in unit DPJ10.

CREATE

Makes a new file or directory.

Format

CREATE*[/switches]* **pathname** *[resolution-pathname]*

The CREATE command produces a new file or directory with the name cited in **pathname**. If the file is a directory, you can then add files to it.

The CREATE command with a /LINK switch creates a file connection. The link file, named in **pathname**, is a very simple name that points to a file with a longer, more complicated name, specified in *resolution-pathname.*

With the /I switch on the command, you can write text into a new nondirectory file. After you enter the command, the CLI displays two right parentheses as a prompt for input. Type in each text line, and close the file with a third right parenthesis next to the CLI)) prompt.

With the /DIRECTORY switch, the command creates a directory subordinate to the current directory with the name cited in **pathname**. If you want to limit the directory's capacity, include a /MAXSIZE= switch on the command. This creates a control point directory. (Be careful when you create a control point directory (CPD). If you run out of directory space while you're in a text editor, the editor won't be able to write the new file to disk and you'll lose all your edits.)

Why Use It?

The CREATE command allows you to establish directories to organize your files. It is also a simple method of creating short text files or macros. Or, you can create a link file to save you keystrokes, because you can type a simple filename that calls a longer, more complicated pathname.

Switches

/DIRECTORY	Creates a directory file.
/I	Creates a file and opens it for input from the terminal.
/LINK	Creates a link, called **pathname**, to a file named in *resolution-pathname.*
/MAXSIZE=	Creates a control point directory with a maximum size of n disk blocks. A disk block holds 512 bytes. You can assign as many disk blocks as you want, as long as you don't exceed the total disk space the system manager allocated to you. You can change the MAXSIZE figure with the SPACE command.

Examples

```
) create/dir Pascal ↓
```

This command creates a new directory called PASCAL.

```
) create/i fas.cli ↓
)) write The files in [!dir] are ↓
)) files/as/s ↓
)) ) ↓
)
```

The preceding command creates a CLI macro called FAS.CLI that displays the working directory name and then lists its files alphabetically.

```
) create/link mortgage :udd:Alexis:Pascal:mortgage  )
```

This command creates a link file called MORTGAGE in the working directory that resolves to a mortgage program in directory :UDD:ALEXIS:PASCAL.

DATE

Displays the current date.

Format

DATE

The DATE command displays the current system date.

The pseudomacro !DATE is a CLI construct, designed to help you write CLI macros. !DATE expands to the system date, as shown in the first example.

Why Use It?

The DATE command is handy when you forget what day it is. The !DATE pseudomacro helps set conditions for the FILESTATUS, DUMP, LOAD, and MOVE commands.

Switches

/L=pathname Writes CLI output to the file specified instead of to your screen.

Examples

```
) date ↵
14-DEC-85
```

This example shows a system date of December 14, 1985.

```
) cre/i  today.cli ↵
)) write Today's files are  ↵
)) files/as/s/after/tlm=[!date]  ↵
)) ) ↵
```

When you execute the macro TODAY.CLI, it will display a message and an alphabetical list of only those files created or modified that day. With the DUMP command, you could use a variation of the TODAY macro to copy only those files created or modified that day to tape or diskette.

DEFACL

Sets or displays the default Access Control List.

Format

DEFACL*[/switches] [username,privilege] ...*

The DEFACL command sets or displays the default ACL, the access automatically assigned to your new files and directories. Usually the default is USERNAME,OWARE. The system manager sets up the default ACL in a system log-on macro. You can set your own default ACL in a personal log-on macro.

Without an argument the DEFACL command displays the current default ACL. You can set a new default by including one or more *username,privilege* arguments with the command. ACL privileges include five types of access: Owner, Write, Append, Read, and Execute. The ACL command, described earlier, gives a detailed explanation of each access privilege.

Why Use It?

Sometimes you'll want to provide limited access to files or directories. By resetting the default ACL, you can automatically grant access to others. For example, you might want to assign Execute access to all files and directories (allowing others to use the directory name in a pathname or execute a program file). And you might want to give colleagues Read access to files and directories, so they can use the FILESTATUS command or see the contents of a file. You rarely want to give Write access, because people could delete, rename, or modify files unconditionally.

Many systems have a logon macro that's executed for each user when he or she logs on. If this macro is present in your initial user directory, you can use a text editor to insert a DEFACL command in this macro. Then the default ACL you specify will be assigned automatically in the future.

Switches

/D Return to the system default ACL, USERNAME, OWARE.

Examples

```
) defacl ⤸
ALEXIS,OWARE
```

The preceding command displays the current default: USERNAME,OWARE.

```
) defacl Alexis,oware project+,re +,e ⤸
```

This command maintains Alexis as owner of all new files and directories, but extends Read and Execute privileges to any user whose username begins with PROJECT. Furthermore, it gives all (+) system users Execute access to new files and directories.

For as long as this default ACL is assigned to files and directories, all system users can use these directory names in a pathname or execute the program files. Furthermore, users whose usernames begin with PROJECT will have Read access to these files and directories, allowing them to see the contents of files or get a complete listing of files within a directory.

DELETE

Eliminates one or more files.

Format

DELETE*[/switches]* pathname *[pathname]* ...

The DELETE command erases directory and nondirectory files, provided you have Owner access to a nondirectory file or write access to the parent directory. The system will not delete a directory that has subordinate directories, unless you use the pathname template character #.

You can use filename templates with the DELETE command arguments.

To protect files from deletion, use the PERMANENCE command.

Why Use It?

As you work on the system, you'll create many files that consume valuable disk space. Some of these files become obsolete or redundant. The DELETE command allows you to remove unnecessary files or directories. You can also copy files to tape or diskette for safekeeping, and then use the DELETE command to free disk space.

Switches

/C Makes you confirm the files or directories before the system deletes them. Directs the CLI to display each file slated for deletion, and wait for your approval. If you answer Y, the CLI will delete the file; if you type any other character, the CLI will not delete it.

/V Verifies the deletion: displays the name of each deleted file.

Examples

```
) del/v  fix ↵
DELETED   fix
) del/v  subdir:fix ↵
DELETED   subdir:fix
```

These DELETE commands erased files named FIX in the working directory and in directory SUBDIR.

```
) del/v/c   test* ↵
test1?      y ↵
DELETED    test1
test2?      y ↵
DELETED    test2
test3?      n ↵
FILE NOT DELETED
test4?      ↵
FILE NOT DELETED
```

The DELETE command directs the CLI to delete all files in the working directory whose name begins with TEST and is followed by any single character except a period. The /V switch directs the CLI to verify each deletion; the /C switch stipulates a confirmation is necessary for deletion. Because of the /C switch, the CLI displays each file matching the template and waits for confirmation. In the dialog, the user deleted two files and retained two others.

DIRECTORY

Displays or changes the working directory.

Format

DIRECTORY*[/switches] [directory-pathname]*

The DIRECTORY command without an argument displays a complete pathname to the current, working directory. When you include *directory-pathname,* the DIRECTORY command changes your working directory to the one specified. To move to a subordinate directory, provide *directory-pathname* from the working directory. To move to a superior directory, use the caret ^ (SHIFT-6) or a full pathname from the root (:).

When you include the /I switch, the DIRECTORY command returns you to your initial (username) directory, or one of its subordinate directories.

Why Use It?

As you work on the system, the DIRECTORY command will help you verify your location. The command also allows you to change your working directory, since it's easier to work within the directory that holds the files of interest than to type long pathnames to these files.

Switches

/I Sets the working directory to the initial user directory. If you include *directory-pathname,* it relocates you to that directory.

Examples

```
) dir ↓
:UDD:ALEXIS
) dir learning ↓
) dir ↓
:UDD:ALEXIS:LEARNING
) dir ^ ↓
) dir ↓
:UDD:ALEXIS
```

The first DIRECTORY command returns the working directory name, :UDD:ALEXIS. The second command makes a subordinate directory, LEARNING, the working directory. And the DIR ^ command makes the superior directory, :UDD:ALEXIS, the working directory.

DUMP

Copies one or more files to a dump file.

Format

DUMP*[/switches]* **dumpfile** *[source-pathname] ...*

The DUMP command copies all files named in *source-pathname* to **dumpfile**. Most often, **dumpfile** is a tape or diskette, but it can be a disk file.

If you omit *source-pathname,* the DUMP command copies *all* files (or *all* those selected by date switches) in the working directory and subordinate directories. If you include any *source-pathname* without specifying a directory name, the command copies only those files in the working directory.

Although the COPY command can also write files to a backup medium, the DUMP command is preferable, because it does the following:

- Dumps files from a directory and subordinate directories with only one command.
- Copies each file under its own pathname, with important information like creation date and ACL.
- Allows template characters and has date switches, allowing you to dump files selectively.
- Allows you to restore all dumped files with one LOAD command.

Data General tape units under AOS/VS have the device names @MTxn, where "x" is B, C, D or J, and where "n" is the number dialed on the unit thumbwheel. The tape file number follows @MTxn, and applies to the tape you are using. You write data to sequential files on each tape, with files numbered from 0 to 99. Therefore, typical tape filenames are @MTB0:0 or @MTC1:0. See the AOS/VS session in Chapter 2 for a detailed explanation of tape backup.

Why Use It?

Periodically, you should copy important files to tape or diskette. Then, when you need the files, you can restore them with the LOAD command. The DUMP command will also make copies of your files for other people, and will cluster related files into a single disk file.

Switches

/AFTER/TLM=dd-mmm-yy	Dumps all files created or modified on or after day dd in month mmm in year yy. The argument dd must be a 1 or 2-digit number; mmm must be the 3-letter abbreviation for the month; yy must be a 2-digit number.
/L	Lists pathnames dumped to the list file (L).
/L=@LPT	Lists pathnames dumped to the line printer.
/V	Displays (verifies) on the terminal the names of all files dumped.

Examples

```
) dump/v  @MTB0:0 ↓
```

This command copies all files in the working directory and subordinate directories to file 0 of the magnetic tape mounted on tape unit MTB0. It also verifies each pathname dumped. Files on magnetic tape proceed sequentially: first you dump to file 0, then file 1, and so on. The command DUMP @MTB0 would have the same effect, defaulting to file 0.

```
) dump/v/l=@lpt  @mtc0:1    -.f77  -.sr ↓
```

This command copies all FORTRAN 77 and assembly language source files *in the working directory* to the second file of the tape on unit MTC0. It sends a list of dumped files to the line printer.

```
) dir/i ↓
) dump/v/after/tlm=9-JAN-86/v/l=@lpt  @mtj0:0 ↓
```

This command dumps all files in and below the user directory that were created or modified after January 9, 1986. The files are dumped to file 0 of the tape on unit MTJ0, and their names are listed by the line printer. (See the DATE command for a macro that you can adapt for this kind of backup.)

FILESTATUS

Lists filenames and statistics.

Format

FILESTATUS*//switches] [pathname] ...*

The FILESTATUS command displays information on files in any directory. If you omit *pathname*, the command describes all files in the working directory. With a directory name included in *pathname*, the command lists all files within that directory; it will also display a header describing the directory.

You may use templates in arguments to the FILESTATUS command.

Why Use It?

The FILESTATUS command is your primary source of information on files. You'll probably use it more often than any other command to verify the existence of a file, check its name, or gain information about its size and creation date.

Switches

/AFTER/TLM=dd-mmm-yy	Includes all files created or modified on or after day-month-year (dd-mmm-yy.) The dd must be a 1 or 2-digit number; mmm must be the 3-letter abbreviation for the month; and yy must be a 2-digit number.
/ASSORTMENT	Includes an assortment of information: the filename, type, date and time created, and its size in bytes.
/BEFORE/TLM=dd-mmm-yy	Includes files created or modified before the given date. Has the same format as /AFTER.
/L	Sends filenames to the list file.
/L=@LPT	Sends filenames to the line printer.
/PERMANENCE	Describe the permanence attribute: ***PERM*** if on; nothing if off.
/SORT	Sorts filenames alphabetically.
/TYPE=DIR	Include only directory files.

Examples

```
) files  temp+ ↓
```

The preceding command lists on the screen the name of every file in the working directory that begins with the characters TEMP.

```
) files/as/s  +.f77 ↓
```

This command requests an assortment of information about all files in the working directory whose names end in .F77. The /S switch sorts the filenames alphabetically.

```
) files/as/ty=dir/ty=cpd  ↓
```

Here, the CLI should display the names, and other assorted information, on all directories within the working directory.

```
) dir/i ↓
) f/as/s  # ↓
```

This command requests an assortment of information, sorted alphabetically by filename, of all files in the working directory and subordinate directories. For examples with date switches, see the DATE or DUMP command.

```
) f/as/s :udd:Alexis:#  ↓
```

The preceding command requests a listing of files in Alexis's initial directory and all subdirectories.

HELP

Displays information about CLI commands and topics.

Format

HELP*[/switches]* [**COMMANDS* | *command* | *letters*] ...

AOS/VS has extensive, on-line Help messages. They tend to be self-explanatory and usually point you to further information.

The HELP command, without an argument, displays a list of items that relate to the Data General software on your system. With arguments, the command displays more focused information. General types of arguments include

**COMMANDS*	For a listing of all CLI commands, type HELP *COMMANDS.
command	For information on a specific command, type HELPV and the name of the command.
letters	For a listing of all CLI commands that begin with a given character string, type HELP and the first few letters in that string.

If a message is too long for your screen, use the CTRL-S CTRL-Q sequence to read it segment by segment. Or, turn on page mode with the CHARACTERISTICS command.

Why Use It?

The HELP command can be extremely useful when you can't remember the name of a command, its effect, or its syntax.

Switches

/L	Writes the Help message to the current list file.
/L=@LPT	Writes the Help message to the line printer.
/V	Or simply type HELPV. Provides a detailed (verbose) description. Without the V or the /V switch, you'll see a brief description of the command syntax.

Examples

```
) help l ↵
L- POSSIBILITIES   ARE:
LEVEL
LISTFILE
LOAD
.
.
.

) help list ↵
LISTFILE - ACCEPTS ARGUMENT(S)
SWITCHES: /1= /2= /L(=) /G /K /P
FOR MORE HELP, TYPE
HELP/V    LIST

) helpv list ↵
                        L I S T F I L E   C o m m a n d
      Format:
      LISTFILE   [pathname]

      Displays (without the pathname argument) the current listfile
      setting, or sets it to the pathname specified .  If you have not
      set the listfile, the CLI returns @LIST, the generic listfile,
      since there is no listfile setting, except in batch mode.  When
      ....
```

The first HELP command displays the names of all CLI commands that begin with the letter L. The second HELP command displays a terse message about the LISTFILE command, whereas the last command provides a very detailed description of it.

LISTFILE

Displays or sets the generic @LIST filename.

Format

LISTFILE*[/switches] [pathname]*

The list file, @LIST, is one of the AOS/VS generic files, which means it is really a pointer to another file, which you must create at runtime with the LISTFILE command. The purpose of a generic file is device independence — you can code programs to write to the generic file @LIST, and at runtime, assign the generic file to a real file, such as @CONSOLE or a disk file.

With an argument, the LISTFILE command sets @LIST to a specific file. Without an argument, the LISTFILE command displays the current file. (If it's @LIST, then it isn't set.)

Both user programs and the CLI access the file pointed to by @LIST. (But if a program tries to open @LIST and you have forgotten to set the LISTFILE, the program will stop with a FILE DOES NOT EXIST message.)

If you specify a disk filename and the file doesn't exist, LISTFILE will create it. If the filename *does* exist, LISTFILE will open it for *appending*, which means that new output will be added to the end of the file.

Why Use It?

The LISTFILE command is most useful with programs that you write, compile, and link into program files. Instead of specifying a literal filename in your source code, you can specify @LIST. Then, from the CLI, you can set and change the list files at will; you don't need to change the source, recompile, and relink.

You can also use the LISTFILE command to get a hardcopy listing of information you'd ordinarily see on your screen — for example, FILESTATUS.

Switches

/G Sets the list file to @LIST.

Examples

```
) list ⤸
@LIST
) list @console ⤸
) xeq mortgage ⤸
... (Output displayed on screen) ...
) list mortgage.out ⤸
) xeq mortgage ⤸
... (Output to file mortgage.out) ...
```

The first LISTFILE command checks the list file; it is @LIST, which means that it isn't set. The next LISTFILE command makes the list file @CONSOLE, so that when the MORTGAGE program executes, the output goes to the terminal. The output looks all right, so the user sets the list file to a disk file named MORTGAGE.OUT, and the output goes to MORTGAGE.OUT.

LOAD

Copies one or more files from a dump file to the working directory.

Format

LOAD*[/switches]* **dump-pathname** *[source-pathname] ...*

The LOAD command complements the DUMP command. It copies files from a dump file (**dump-pathname**) into your working directory. These can be files you dumped earlier for safekeeping, or they can be dump files supplied to you by someone else. A dump file can be a tape file, a diskette, or a disk file.

The CLI tries to load all files in **dump-pathname**, unless you include *source-pathname*. In this case, it loads only the specified file(s). Any directory structure within *source-pathname* is maintained.

If a file within the working directory has the same pathname as a file in **dump-pathname**, the CLI will signal an error, and not load the file — unless the dumped file is newer and you use the /RECENT switch.

You may use templates in arguments to the LOAD command.

Why Use It?

The LOAD command is the easiest way to restore one or more files from backup media to disk. With the /N switch, you have an easy way to read what's on a tape or diskette.

Switches

L	Writes to the list file the names of all files loaded.
/L=@LPT	Writes the names of all files loaded to the line printer.
/N	Displays the pathnames in **dump-pathname** without loading them; thus the switch identifies files in the dump file for you.
/RECENT	Deletes the existing file and loads the newer one, if a filename in **dump-filename** has a more recent creation date than the same filename in the working directory.
/V	Displays the names of all files loaded; the system ignores the /V switch if you also specify /L.

Examples

```
) load/v @mtb0:0 ↵
```

This command directs the system to load into the working directory all files and directories in the first tape file (file 0); the magnetic tape is mounted on unit 0 of the MTB tape drive. The command LOAD/V @MTB0 has the same effect as the example since it defaults to file 0.

```
) load/v @mtd0:1  -.f77  -.sr ↵
```

This command directs the CLI to load into the working directory every FORTRAN 77 and assembly language source file from tape file 1 on unit MTD0.

```
) load/n @mtc0:1 ↵
```

The preceding command directs the CLI to read — but not load — the contents of tape file 1 and display the filename(s) on the screen.

LOGFILE

Displays or sets the current log file.

Format

LOGFILE*[/switches] [pathname]*

A log file records all CLI dialog directed to or from your terminal.

The LOGFILE command, without arguments, displays the current log filename, if there is one. The command with *pathname*, directs the CLI to open the file named in *pathname* and append terminal dialog to it. The CLI creates *pathname* if it doesn't exit. If, by chance, there's an active log file, and you enter another LOGFILE command with a different *pathname*, the CLI closes the old log file and opens a new one, as specified in *pathname*. Each log file remains in the directory where you started it, but you can only have one log file open at any time.

When you start a log file, it records dialog until you do one of the following:

- Type LOGFILE/K.
- Start a new log file.
- Log off the system.

You can restart any log file by specifying its name with the LOGFILE command, but the system won't automatically open a log file for you.

All CLI dialog, except output from the TYPE and HELP/V commands, is recorded in the current log file. Terminal input and output with programs, such as text editors or compilers, is not recorded in the log file.

Why Use It?

A log file is a good way to monitor terminal input and output or trace problems in program development. In addition, a log file will record the output resulting from any command issued with a /VERIFY switch, so it can be a faster method of collecting data than directing output to the printer with the L=@LPT switch.

Switches

/K Terminates (kills) the log file, stops recording, and closes the file.

/V Displays the pathname of the current log file. (Knowing the pathname is useful when you're setting the logfile.)

Examples

```
) logfile ↓
) logfile/v session.log ↓
:UDD:ALEXIS:LEARNING:SESSION.LOG
... (Dialog with the CLI) ...
) logfile/k ↓
) qprint session.log ↓
```

The first command checks to see if there's a log file open, but there is isn't. The next command opens a log file SESSION.LOG in the working directory and starts recording terminal dialog. The /V switch directs the CLI to confirm its pathname. CLI dialog follows. Then Alexis closes the log file with the /K switch on the command, and prints the file SESSION.LOG.

MOVE

Copies one or more files to a different directory.

Format

MOVE*[/switches]* **destination-directory** *[source-pathname] ...*

The MOVE command places a copy of one or more files in **destination-directory**. The *source-pathname* must either be in the working directory or one subordinate to it. The command can copy both individual files or directory structures. If you omit *source-pathname*, the system copies the files in the entire directory structure, from the working directory down to the destination directory. To move any files, you must have Append or Write access to the directory specified in **destination-directory**.

You can use templates as part of *source-pathname*.

Why Use It?

From time to time, you may want to reorganize files or transport files from one computer system to another.

The MOVE command retains the file's original name, creation date, and so on; thus, it identifies the file better than the COPY command does. The MOVE command also allows multiple source pathnames and template characters, whereas the COPY command does not.

To conserve disk space, you might want to delete the files from the original directory after the move. Otherwise, you'll have duplicate files in different directories.

Switches

/AFTER/TLM=dd-mmm-yy	Moves all files created or modified on or after dd-mmm-yy, where dd is day, mmm is month, and yy is year. The argument dd must be a 1- or 2-digit number; mmm must be the 3-letter abbreviation for the month; yy must be a 2-digit number.
/BEFORE/TLM=dd-mmm-yy	Moves files created or modified before dd-mmm-yy.
/L	Lists pathnames moved to the list file.
/L=@LPT	Lists files moved to the line printer.
/RECENT	Deletes the older file; moves in a copy of the newer file, if the source filename is more recent than a file with the same name in **destination-directory**.
/V	Verifies filenames moved. (/V is ignored if you also use the /L switch.)

MOVE (continued)

Examples

```
) move/v FORTRAN +.f77  ↓
MORTGAGE.F77
COUNT.F77
ERROR: FILE ... EXISTS: FORTRAN:MPROG.F77
) move/v/r FORTRAN +.f77 ↓
MPROG.F77
```

The first MOVE command tells the system to move every file in the working directory ending in .F77 to the subordinate directory FORTRAN. The CLI moves and confirms two files, but reports an error on MPROG.F77. There's already a copy of MPROG.F77 in the directory FORTRAN.

Adding the /RECENT switch to the MOVE command tells the system to place the newer version of the file MPROG.F77 in FORTRAN. The command *does* move MPROG.F77 into directory FORTRAN because the file in the working directory is more recent than the existing copy in FORTRAN.

PERMANENCE

Displays or sets the Permanence attribute for one or more files.

Format

PERMANENCE*[/switches]* filename *[ON / OFF]*

The Permanence attribute, if on, prevents any directory or nondirectory file from being deleted by any user. (But Permanence will not save a file if its parent directory is not permanent and someone deletes the parent directory.) You must have Owner access to delete a file.

If you omit the argument *ON* or *OFF*, the PERMANENCE command displays the current Permanence setting. With an argument, it sets Permanence on or off. You can use template characters with filename.

Because text editors and other utilities delete files and recreate them as part of their function, the PERMANENCE command *can* interfere with everyday program development. So you may want to set PERMANENCE ON only for directories and selected files.

Why Use It?

It's not hard to delete a file or directory accidentally — especially if you're casual with the DELETE command and template characters. If this happens, and the file hasn't been backed up, it's lost forever. Keeping Permanence on prevents such unpleasant events.

Switches

/V Displays the filename along with the Permanence attribute.

Examples

```
) cre/dir learning  ↵
) perm/v learning  ↵
LEARNING    OFF
) del/v learning ↵
DELETED  LEARNING
) cre/dir  learning ↵
) perm learning on ↵
) perm/v learning ↵
LEARNING  ON
) del/v learning ↵
WARNING:  CANNOT DELETE PERMANENT FILE, FILE learning
```

In this sequence, a user created a directory, checked its Permanence attibute, which was off, and then deleted the directory. The user then recreated it, set Permanence on, and tried to delete the directory again. This time, with Permanence on, the CLI refused to delete it.

QBATCH

Creates and submits a batch job.

Format

QBATCH argument *[argument] ...*

The QBATCH command creates a batch input file, with the CLI commands that you submit as arguments, and passes this file of commands to the CLI. The CLI sets up a new process to execute the job, freeing your terminal for further activity. When the job is complete, the system sends the results to an output file — usually the line printer.

Why Use It?

The QBATCH command allows you to compile programs or sort large files noninteractively, thus freeing your terminal for other tasks. Also, batch jobs can run during slow periods, when system demand is low. You can specify the time to run a batch job with the /AFTER switch.

Switches

/AFTER=date:time	Runs as soon as possible on the specified date and/or time. The date:time is in the form dd-mmm-yy:hh:mmm:ss, (for example, 10-JAN-86:08:43:00) but you can use date or time alone. The default time is midnight. You can also use /AFTER=+6 to mean six hours from now.
/I	Takes the input file contents from subsequent lines typed at the terminal (@INPUT). So the CLI can identify the end of input, *you must close the input by typing a right parenthesis*) *and pressing the NEW LINE key.*
/NOTIFY	Notifies you when the job is complete.
/QOUTPUT=pathname	Sends the output file to the specified pathname. The default is BATCH_OUTPUT, usually the line printer.
/V	Verifies the transaction by displaying the batch input filename.

Example

```
) qbatch/v f77 mortgage.f77  ↵
:UDD:ALEXIS:FORTRAN:?026.CLI.001.JOB  QUEUED, SEQ=n QPRI=n
) qd ↵
BATCH_INPUT    BATCH    OPEN
*7539  D   ALEXIS   :UDD:ALEXIS:FORTRAN:?026.CLI.001.JOB
.
.
.
```

This QBATCH command creates a batch job file, identified by the suffix .JOB, and places it in the BATCH_INPUT queue. The system sets up the batch process, which is the compilation of a FORTRAN program, and executes the FORTRAN compiler. Then the system sends the contents of BATCH_INPUT, along with any compilation errors, to the BATCH_OUTPUT file, which in this example defaults to the line printer. (See Chapter 2 for other examples of the QBATCH command.)

QDISPLAY

Describes jobs in the batch and print queues.

Format

QDISPLAY*[/switches]*

The QDISPLAY command describes the batch and print queues. A queue is a file that stores print and batch requests until the printer and the system are ready to process them. The system lists each print request in the queue; and, generally, the printer completes jobs on a first-come, first-serve basis.

Why Use It?

You will frequently want to know whether the printer is free, or how soon it may be free. Other times, you may want to confirm that a job is complete before you walk to the line printer. QDISPLAY is a convenient command.

Switches

/V Includes the estimated number of pages the file will produce. This number appears under the head LIMIT. The system considers 1,000 characters as a page and adds four pages for headers and trailers; thus, for a 10,000-character file, it would estimate 14 pages.

Examples

```
) qd ↵
BATCH_INPUT     BATCH   OPEN
  7506 A    OP        :UDD:OP:MACROS:DAILY_SWEEP
.
.
.
LPT             PRINT   OPEN
* 7544      ALEXIS    :UDD1:ALEXIS:LTU:7

LQP             PRINT   OPEN
.
.
.

) qd/v ↵
LPT             PRINT   OPEN
  SEQ# PRI   TIME      LIMIT FLGS USERNAME FORMS    PATHNAME
         15-AUG-85
* 7545 127 13:06:12       27      ALEXIS            :UDD:ALEXIS:LEARNING:DRAFT1
  7546 127 13:06:14        4      ALEXIS            :FF
.
.
.
```

The first QDISPLAY command shows a batch job DAILY_SWEEP and a print job called LTU:7, queued for the line printer (LPT). The second command provides a fuller explanation of job requests: Alexis's DRAFT1 is printing at 1:06 p.m., and the system expects it to be about 27 pages. The filename :FF is a form feed macro that advances paper in the line printer, so it's easy to tear jobs off without removing another user's output.

QPRINT

Sends one or more files to a line printer.

Format

QPRINT*[/switches]* pathname *[pathname] ...*

The QPRINT command copies to the printer queue the file you specify in pathname. A queue is essentially a waiting list, a file that stores print and batch requests until the printer and the system are ready to process them. The printer will print a copy of the file as soon as the request is first in the queue.

The output on the line printer usually begins with a header page for each pathname, containing your username, its pathname, and the time and date; a copy of the file follows.

Why Use It?

The QPRINT command is the best way to print a file. It is the *only* way to print without tying up your terminal until the file is printed.

Switches

/COPIES=n	Print n copies of the file.
/FOLDLONGLINES	Do not truncate lines longer than the printer width, but continue them onto the the next line of the listing. However, the folded lines may be difficult to read.
/NOTIFY	Tells the system to send a message to the terminal when the print job completes.

Examples

```
) qpr file1 ↵
QUEUED,    SEQ=92
```

This command submits a print job: the line printer will print FILE1, and preface it with a header page.

```
) qpr/notify mortgage.output ↵
QUEUED, SEQ=724, QPRI=127

FROM PID 3:  (EXEC) @LPB COMPLETED :UDD:ALEXIS:MORTGAGE.OUTPUT, SEQ=724
```

The preceding command submits a job to the print queue: MORTGAGE.OUTPUT. The /NOTIFY switch directs the CLI to send a message to the terminal when the job is complete.

RENAME

Assigns a new name to a file.

Format

RENAME old-filename new-filename

The RENAME command gives a new filename to a file specified in old-filename.

Why Use It?

The command is handy when you want to save an initial version of a file. AOS/VS doesn't label versions of files; instead it replaces the original with the modified file. For example, the SED text editor keeps the most recently modified version of a file if you want it to do so, calling it filename.BU, and deletes the older version. Each Data General compiler produces an .OB version of a source file, deleting any existing .OB first. The Link utility produces a .PR version from the .OB, deleting any older .PR file first.

Switches

None.

Examples

```
) copy  myprog.old.f77 myprog.f77  )
) rename myprog.ob myprog.old.ob  )
) rename myprog.pr myprog.old.pr  )
```

In this example the user has a working version of a FORTRAN program named MYPROG.F77 and plans to modify it extensively. The user keeps the working files of MYPROG intact by duplicating the file with the COPY command, naming the new file MYPROG.OLD.F77. Then the user renames the .OB and .PR files. Now it's possible to change the file MYPROG, recompile, and relink it without deleting the old .OB and .PR files.

```
) rename macro.cli supermacro.cli  )
```

This command changes the name of the file MACRO.CLI, which is in the working directory, to SUPERMACRO.CLI.

```
) rename dir2:fritz fritz.old  )
```

This command changes the name of file FRITZ, in the subordinate directory DIR2, to FRITZ.OLD. The file FRITZ.OLD remains in directory DIR2.

SEARCHLIST

Displays or sets your directory search list.

Format

SEARCHLIST*[/switches] [directory-pathname] ...*

A search list is a short list of directories that the system scans if it can't find a file in your working directory. The system manager usually establishes the search list in a log-on macro. You can set a search list in your own log-on macro, in which case it becomes unique — and private — information.

If you omit arguments, the SEARCHLIST command displays your current search list. With arguments, the system sets a new search list. Each *directory-pathname* argument must be a full pathname, starting from the root (:) and ending in a directory name. If you specify more than one pathname, separate the pathnames with a comma. To append a search list, include the pseudomacro !SEARCHLIST, which maintains the original searchlist, while other directories in the command line are added to it.

A search list applies to most CLI commands that involve files, such as XEQ, COPY, and TYPE, and to CLI macros. (It doesn't apply to the DELETE, DUMP, MOVE, or FILESTATUS commands.) When you type a CLI command, the system searches the working directory first and, not finding the file, scans the search list beginning to end, searching the directories specified. The search ends when the CLI finds the first file with the name you supplied, or when it can't find any file by that name.

The CLI will not scan the entire search list if you precede a filename with one of the following characters:

Symbol	Meaning
= (equals sign)	It looks only in the working directory.
@ (commercial "at" sign)	It looks only in the peripherals directory: e.g., @LPT, @MTB0, @CONSOLE.
^ (caret)	It scans only the superior directory.

Why Use It?

With an accurate search list you avoid long pathnames, thus saving many keystrokes. A search list also allows you to execute system utilities without even knowing their full pathnames.

Many systems have a logon macro that's executed for each user when he or she logs on. If this macro is present in your initial user directory, you can use a text editor to insert a SEARCHLIST command in this macro. Then the default search list you specify will be assigned automatically in the future.

Switches

/K Kills or deletes the current search list, if any exists.

Examples

```
) supermacro ⤵
ERROR: NOT A COMMAND OR MACRO, supermacro

) search ⤵
:UTIL, :
) sea [!search] :udd:Alexis:macros ⤵
) sea ⤵
SEA :UTIL, :, :UDD:ALEXIS:MACROS
) supermacro ⤵
... (The macro executes) ...
```

First Alexis tries to execute a macro, knowing it exists; but the CLI can't find it. So Alexis checks the search list and finds that directory :UDD:ALEXIS:MACROS, where SUPERMACRO resides, is not there. Alexis sets the search list to include MACROS, using the pseudomacro !SEARCHLIST to include the current search list. When Alexis tries SUPERMACRO again, it executes.

SPACE

Sets or displays the amount of disk space in a control-point directory.

Format

SPACE *[:UDD:user-directory] [new-max-size]*

As a user, you have a fixed amount of disk storage space for all your directories and files. The system manager establishes the amount when seting up your account.

The SPACE command displays how much space you have (MAX), how much is occupied by files (CUR), and how much remains free (REM). It calculates space in units of disk blocks, each block holding 512 bytes (characters).

The SPACE command works only with *control point directories*. All initial user directories and the root (:) are control point directories, so the SPACE command works with any of these. You can create control point directories, subordinate to your intial directory, with the /MAXSIZE= switch on the DIRECTORY command. Omit arguments to the SPACE command if the working directory is *user-directory*.

By including *new-max-size* on the SPACE command line you can change the size of any control point directories subordinate to your initial directory. The total size of your directories remains limited by the number of disk blocks that the system manager assigned to you when establishing your account.

Why Use It?

The SPACE command keeps you up-to-date on the amount of disk space you have. You can use this information to decide when to clean up your directories — that is, copy files to tape and delete obsolete files and directories. You can also use the SPACE command to enlarge the control point directories you've created.

Switches

None.

Examples

```
) dir learning ↓
) space ↓
WARNING: ...NOT A CONTROL POINT DIRECTORY...
) dir :udd:Alexis ↓
) space ↓
MAX 15000, CUR 39, REM, 14961
```

In this example, Alexis typed the SPACE command from an subordinate directory and received an error message. Then, after making the initial user directory the working directory, Alexis tried the command again and got results. Directory :UDD:ALEXIS has a total work area of 15,000 disk blocks: 39 are currently occupied by files and 14,961 blocks remain free.

TIME

Displays the current system time.

Format

TIME

The TIME command displays the current system time, which is based on a 24-hour clock. Midnight is 00:00; 12 noon is 12:00; and 6:00 pm is 18:00.

Why Use It?

The TIME command is simply a convenience. The pseudomacro !TIME is useful in macros because it expands to the current time.

Switches

None.

Examples

```
) time ↓
14:17:06
```

The current time is 17 minutes and 6 seconds after 2:00 p.m.

```
) write it is now [!time] ↓
IT IS NOW 14:53:26
```

The pseudomacro expands to the current time.

TYPE

Copies one or more files to your screen.

Format

TYPE pathname *[pathname] ...*

The TYPE command displays on the terminal screen the file named in pathname.

The TYPE command works properly only for text (ASCII) files. To display a non-ASCII file, one ending in .PR, .OB, .ST, or .LB, use the DISPLAY utility, described in the *Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS).*

As before, you can use CTRL-S and CTRL-Q to suspend and resume the display of text on your screen. You can also put your terminal in page mode with the CHARACTERISTICS command. To discard output — a time saver on a printing console — use CTRL-O.

Why Use It?

The TYPE command is the fastest and easiest way to see what's in a file. You can use it on CLI macros and other text files.

Switches

None.

Examples

```
) type file2 ⤸
... (Text of file2) ...
) ty :util:paru+ ⤸
... (Text of the AOS/VS USER PARAMETER file) ...
```

These commands display FILE2 and the AOS/VS USER PARAMETER file on the terminal screen.

WHO

Displays the username of a process.

Format

WHO *[pid-number] ...*

The WHO command displays the username and program file of a process. Without an argument it describes your own process: its Process ID number (PID), your username, process name, and the program you're running. When you supply *pid-number*, the WHO command describes that process in the same terms: username, process name, program name.

Why Use It?

The WHO command is useful in identifying which process you're running, or identifying who sent you a message with the SEND command. See the *Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS)* for a description of SEND.

Switches

None.

Examples

```
) who )
PID:   14 ALEXIS          034              :UTIL:SED.PR
) who 25 )
PID:   25 FRAN            CON18            :CLI.PR
```

The first command describes Alexis's process, which is PID 14, running the SED text editor. The second command reports that PID 25 is user FRAN, who's running a CLI program.

WRITE

Displays arguments.

Format

WRITE*[/switches] [argument] ...*

The WRITE command either displays *argument* on the terminal screen or writes the argument to a file. The *argument* can be a text string and/or one or more of the CLI's pseudomacros. If you omit arguments, the WRITE command displays a blank line.

Generally, avoid using CLI punctuation, such as a semicolon (;), *within* arguments to the WRITE command, because the CLI will interpret them.

If you use the /L=pathname switch, the WRITE command will *append* text to a disk file.

Why Use It?

The WRITE command can help you document macros or log files, and add text to disk files.

Switches

/L=pathname Writes argument to a file named in pathname, instead of the terminal screen.

Examples

```
) cre/i ?.cli ↓
)) write System users at [!time],[!date] are ↓
)) write ↓
)) who/2=ignore <0,1,2,3,4,5> <0,1,2,3,4,5,6,7,8,9> ↓
)) write ↓
)) write Your username is [!username] and ↓
)) write Your process ID is [!pid]. ↓
)) write Your working directory is [!dir]. ↓
)) write Your searchlist is [!sea]. ↓
)) write Current space in :udd:[!username] is ↓
)) space :udd:[!username] ↓
)) ) ↓
```

This example shows a macro, ?.CLI, that you can execute by typing ? ↓. The macro displays information on users with process IDs 0 through 59, and then provides additional information on your own process. The bracketed pseudomacros are further described in the Help messages and in the *Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS)*. (Since the macro is rather lengthy, you should probably use a text editor to type it in. The next chapter describes the SED text editor.)

XEQ

Executes a program.

Format

XEQ pathname *[argument]*

The XEQ command runs the program file named in **pathname**. (A program file is one that has been compiled or assembled, and built into an executable program; its name always ends with a .PR suffix.) You can omit the .PR suffix from **pathname**.

The optional *argument* is typically the name of a file that you want the program to access and process. It may also be an argument to the program that you want to run. Generally, you supply a filename argument when you execute system utility programs such as the SED text editor or the Link utility.

A program that you write and build has the filename you gave the source file, with the appropriate three-character suffix replaced by .PR. For example, a FORTRAN 77 source file, MPROG.F77, would compile into the object file, MPROG.OB, which in turn would produce program file, MPROG.PR. You can run it as MYPROG.

However, if you gave MPROG an unconventional suffix, such as MPROG.FT, the compiled object file would be MPROG.FT.OB, and the program file would be MPROG.FT.PR. You would run it as MPROG.FT.

Why Use It?

Use the XEQ command to run any program file — a compiler, text editor, or your own program. Sometimes system macros exist for programs, so all you have to do is type the product name. If you receive an error message from the XEQ command, try omitting XEQ from the command line.

An more versatile (and complex) alternative to the XEQ command is the PROCESS command, described in the *Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS)*.

Switches

None.

Examples

```
) xeq sed mortgage.f77 ↓
Do you want MORTGAGE.F77 to be created? Y ↓
*... (SED editing session) ...
) f77 mortgage.f77 ↓
) f77link mortgage.ob ↓
) xeq mortgage ↓
```

In this example, the user entered the XEQ command to run the SED text editor, and in SED wrote a FORTRAN 77 source program. The user entered CLI macros (the second and third command lines) to compile and link the program. Finally, the last command line executed the program itself.

What Next?

Now you may want to learn a text editor. If so, continue to Chapter 4 to learn about SED, a screen-oriented text editor, or to Chapter 5 to learn SPEED, a character-oriented text editor. Later chapters introduce AOS/VS programming languages and the Sort/Merge utility. Chapter 16 guides you to other documentation on AOS/VS products.

End of Chapter

Chapter 4
Writing with the SED Text Editor

This chapter shows you how to use the SED text editor. As with the CLI, the best way to learn SED is to use it. So we describe essential features of SED; then you start using it in a working session, similar to the session with AOS/VS in Chapter 2.

Table 4-1 summarizes the commands that the chapter covers. For a complete description of the editor, see the *SED Text Editor User's Manual (AOS and AOS/VS)*.

Using SED

To create and edit a file with the SED editor, you must have Append and Write access to your working directory. (You always have this access if you're executing SED from your user directory.)

The SED prompt is an asterisk (*), appearing at the top of your terminal screen. You will type SED commands next to the * prompt. In this chapter, the symbol (ESC) means that you press the ESC key. The ESC key changes the command mode; however, it does not display any character on your screen when you press it.

Cursor and Case of Characters

The *cursor* on a terminal screen indicates the current position on a line. The cursor is either a box, superimposed on the current character, or an underscore that blinks beneath the current character.

The SED editor is not, by default, case sensitive. So, if your terminal has both lowercase and uppercase characters, you can use either for both commands and text. All compilers accept either case in text strings, except for FORTRAN 5 and COBOL, which require *statements* to be in uppercase. In the session, we use lowercase.

Locating a Template

Across the top of your terminal keyboard is a row of function keys. A cutout called a template fits over them, labeling which keys represent which commands. (See Figure 4-1 for an example.) If you don't have a template, ask your system manager for one, since text editing is much faster with function keys.

If You Make a Mistake

The SED editor handles errors well and has an array of informative error messages. As you move through the session, it's okay to make your own mistakes. The main point is to learn the way SED works.

SED Command Summary

Table 4-1 summarizes the SED commands we'll use in the session. Then we briefly describe the SED function keys, pictured in Figure 4-1.

Table 4-1. The SED Text Editor Commands

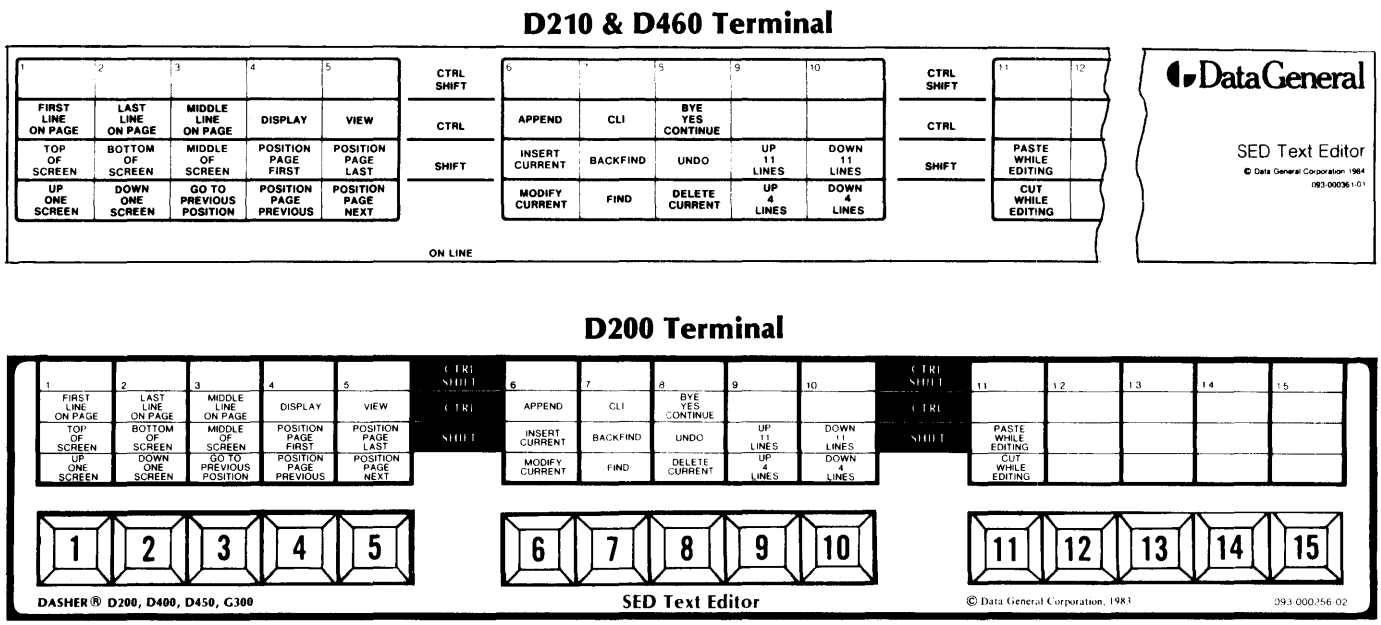
Command and Its Format	Task	Example
XEQ SED *[pathname]* or SED *[pathname]*	Start the editor.	) xeq sed source-file ↵) sed source-file ↵
APPEND FROM *[pathname]*	Add text to the file.	* append from file1 ↵
BYE	Close and update file and return to the CLI.	* bye ↵
DELETE *[range]*	Delete one or more lines.	* delete 18 20 ↵
DUPLICATE *[range]* {BEFORE address | AFTER address | ONTO pathname}	Copy text within the file or onto another file.	* dup 13 bef last ↵ * dup 85 aft 60 ↵ * dup 3 onto file1 ↵
FIND string *[range]*	Search for a word or phrase.	* Find "thea" all ↵
HELP *[keyword]*	Obtain help.	* help ↵
INSERT *[address]*	Insert new lines of text.	* insert 16 ↵
JOIN *[address]*	Remove a page break.	* join 2 ↵
LIST *[range]*	Display all or part of a file.	* list 1 20 ↵
MODIFY *[address]*	Change one or more lines.	* mod 2 ↵
MOVE *[range]* {BEFORE address | AFTER address | ONTO pathname}	Rearrange text within the file or append to another file.	* move 23 bef 6 ↵ * move 50 56 aft 88 ↵ * mov 13 onto file1 ↵
POSITION address	Change position.	* po pa 2 ↵
SAVE	Update a file on disk.	* save ↵
SPLIT	Make a new page.	* split ↵
SUBSTITUTE string FOR string *[range]*	Replace each occurrence of a word or phrase with another one.	* sub y for i ↵
VIEW	Display a range of lines around the current line.	* view ↵

The SED Function Keys

SED function keys, the top row of keys on your terminal, can reduce keystrokes and make editing a lot easier. Each function key represents a command. In combination with other keys — like SHIFT or CTRL — each key can represent three different commands.

In the session, we'll describe the commands in such a way that you can either use the function keys or type out commands on the SED command line. Since it's faster to use function keys, locate a template if at all possible and place it over the top row of keys.

Figure 4-1 shows SED templates. Note that many function keys relate to your position within a file. And the balance are standard text editing commands — to add and modify text within a file. We'll use these keys in the session. In SED you can define your own function keys, and the SED User's manual explains how.



DG-27420

Figure 4-1. SED Templates

A Session with SED

To illustrate each SED command, we'll create and edit a text file named HISTORY. The dialog includes several deliberate errors that you should enter as shown.

Log on the system, if necessary. Ensure that the working directory is your user directory by typing

```
) dir/i  ⤵
```

Starting SED

Execute the SED text editor by typing

```
) xeq sed history  ⤵
```

(If you receive an *ACCESS DENIED, X, SED* error message, omit XEQ from the command line and try again.)

The editor displays its banner, and requests confirmation before creating the file HISTORY. You do want a new file called HISTORY, so enter Y:

```
SED  Rev. n   Input file -   HISTORY
Do you want HISTORY to be created?    y ⤵
```

SED creates the file, and then displays its editing screen with the command line on top:

```
*                                                          page 1 line 1
-----------------------------------------------------------------------
```

Notice the line — it runs across the screen, separating the command field from the text field. The SED prompt * is above this line. Whenever the cursor is next to the prompt, SED is ready to receive a command. On the far right of the command field, SED displays your current position in the file. You have an empty file at this point, so your position is line 1 on page 1.

After you enter a command, SED moves the cursor to the text field, to mark your current line position in the file.

Writing text with SED generally involves a three step procedure:

Step 1: Select the command mode.

Step 2: Add to or edit text within the file, pressing NEW LINE (or an uparrow or downarrow key) to have SED record the new text.

Step 3: Press the ESC key, notifying SED to exit from the current mode, return to the command field, and wait for the next command.

You can select a command by either pressing the appropriate function key or typing the command on the command line.

You can return to the CLI at any time by typing the SED command CLI. If you need to exit from SED, enter the BYE command.

Now, let's get started.

Adding and Displaying Text

SED just created a file named HISTORY, to which we'll add text. We use the APPEND command to add text to a file (even if it's an empty file).

If you have a template, press the CTRL key with the APPEND function key. Or, to enter the command on the command line, type

```
* append ⤸
```

SED positions the cursor below the command line, and waits for you to add text. SED always displays line numbers (unless you tell it not to), so the the text field looks like this:

```
view
*   append                                              page 1 line 1
------------------------------------------------------------------------
1     _
```

The cursor is at the beginning of line 1. Now type the following source lines, as they appear, and press the ESC key to complete the entry. Type

```
1       In the 1850's both Charles Darwin and Alfred Russel Wallace ⤸
2       proposed the theory of evolution.  Each theorist was basically ⤸
3       a naturalest, although he did have a little background in science. ⤸

(ESC)
```

When SED is executing a command, the prompt and command remain at the top of the screen. Here, you're appending text from the terminal. To append text from a disk file, you can type the command APPEND FROM and supply the pathname as an argument.

The current line is the last line appended. You can easily tell the current line because it is *bright*, whereas the surrounding lines are dim. The line numbers followed by tabs are for your convenience; they do not become part of the file.

Now, to display the text in a file, use the LIST command. The command allows you to see a range of text. For example, type

```
* list all  ⤸

1       In the 1850's both Charles Darwin and Alfred Russel Wallace
2       proposed the theory of evolution.  Each theorist was basically
3       a naturalest, although he did have a little background in science.
```

ALL is a modifier that specifies the range — here, all lines in the file. You could also use the argument "1 3" meaning display lines 1 through 3. (This isn't much of a test for the LIST command, but it makes a point.)

To continue, let's complete the source file. Return to the text file with the APPEND command, either by pressing the APPEND function key, or typing

```
* append    ↵

4       Darwin studied in Edinburgh, and Wallace attended ↵
5       surveyor's apprentice.  ↵
(ESC)
```

Now the source file is complete. Let's see how to modify it.

Modifying Text

You'll use the MODIFY command to edit text. Now let's use it to fix the typographical error in line 3. Either press the MODIFY CURRENT key and use cursor control keys to move the cursor to line 3, or, on the command line, type

```
* modify 3 ↵
```

When you modify, SED places the cursor at the beginning of each line. Use the cursor control keys — leftarrow, rightarrow, or HOME to move one or more spaces left or right, or to the beginning of the line again. Also use CTRL-A, CTRL-F, CTRL-B to move the cursor to the end of the line, forward one word, or backward. When you are done, press NEW LINE to inform SED that the line is complete.

The screen editing characters that we used in the AOS/VS session work the same way for SED as for the CLI. You can use them in APPEND, MODIFY, and INSERT mode, as well as on the SED command line. The only difference is CTRL-A: in Modify or Command mode it moves the cursor to the end of the line. Otherwise, it repeats the last line typed. Table 3-1 lists the screen editing keys, and the AOS/VS session in Chapter 2 provides an exercise using them.

Now let's use screen edit keys to correct the typographical error, *naturalest*.

The cursor is in the HOME position, on line 3, right under the A. To advance the cursor to the typing error, press CTRL-F once, then press the rightarrow key seven times.

The CTRL-F moves the cursor under the first letter of *naturalest* ; the rightarrow key moves it to the incorrect character, *e*. Type a lowercase i and press NEW LINE.

```
3       a naturalest, although ...
                 -
```

Type

```
i ↵
```

The "i" overwrites the "e," and pressing the NEW LINE key directs SED to record the corrected line in the file. SED then displays the next line, placing the cursor on the *D* in *Darwin*.

When modifying, you can move through a file several ways: by specifying a position on the command line, using cursor control keys, or using function keys. If you specify no position, SED assumes the current line. SED highlights the specified line. In any case, SED displays lines sequentially, until you press the ESC key or reach the end of the page.

Next, we need to insert missing text in line 4. The cursor is now on line 4 in the HOME position.

```
4       Darwin studied in Edinburgh, and Wallace attended
```

The word "medicine" needs to go before the phrase "in Edinburgh". Press CTRL-F two times; then press CTRL-E, and type **medicine** in lowercase letters; press the space bar, CTRL-E, and the NEW LINE key.

```
CTRL-F  CTRL-F  CTRL-E  medicine  CTRL-E  ↵
```

CTRL-F moved the cursor forward to the word "in". Then you pressed CTRL-E to open the line for a text insert. You typed the text "medicine," spaced once, and ended the insert with CTRL-E. Finally, you pressed NEW LINE to inform SED that the line is complete.

The line now reads,

```
4       Darwin studied medicine in Edinburgh, and Wallace attended
```

Press the ESC key to leave the MODIFY command.

At this point, you've added text to a file with the APPEND command, displayed it with the LIST command, and corrected it with the MODIFY command. You've learned you can issue commands to the SED editor by pressing a function key or typing commands on the command line.

Inserting and Deleting Text

The file HISTORY looks like this:

```
1       In the 1850's both Charles Darwin and Alfred Russel Wallace
2       proposed the theory of evolution.  Each theorist was basically
3       a naturalist, although he did have a little background in science.
4       Darwin studied medicine in Edinburgh, and Wallace attended
5       surveyor's apprentice.
```

There's clearly something missing — the file doesn't explain which school Wallace attended. The INSERT command allows you to insert one or more lines before the specified line; if you omit a line number, it inserts text before the current line.

Let's insert a line between 4 and 5. To do so, either position the cursor on line 5 and press the INSERT CURRENT function key, or type

```
* insert 5 ↵                                              page 1 line 5
-----------------------------------------------------------------------
.
.
.
4       Darwin studied medicine in Edinburgh, and Wallace attended
5       _

*       surveyor's apprentice.
```

Notice that SED moves the text from line 5 down, leaving plenty of room for new text. Enter the missing information on line 5. Type

```
5       the Working Men's Institute in Leicester, where he served as a ↵
```

The INSERT mode continues, with SED displaying updated line numbers, until you press the ESC key. So press

```
(ESC)
```

Now use the LIST command to display the file. Type

```
* list all ⤶
```

```
1       In the 1850's both Charles Darwin and Alfred Russel Wallace
2       proposed the theory of evolution.  Each theorist was basically
3       a naturalist, although he did have a little background in science.
4       Darwin studied medicine in Edinburgh, and Wallace attended
5       the Working Men's Institute in Leicester, where he served as a
6       surveyor's apprentice.
```

In SED you can erase text lines with the DELETE command. It eliminates the line or range of lines that you specify. If you omit line numbers, DELETE deletes the current line.

Let's get rid of line 6. To do so with function keys, first ensure line 6 is the current line (by positioning the cursor or pressing NEW LINE), and then press the DELETE CURRENT key.

Or, type the command

```
* delete 6 ⤶
```

The remaining file has 5 lines. But the paragraph is incomplete, so retrieve the line with the UNDO command.

Press the UNDO function key, or type

```
* undo ⤶
```

And SED restores the line.

The UNDO command is useful when you accidentally delete desired text. (The command will replace the last deletion, but none prior to it.)

Finding Text

Locating text strings is an important part of the editing process. The FIND command searches for a text string, and, if found, highlights the line within the surrounding text.

Enclose a text string with either apostrophes or quotation marks whenever the string contains spaces, case sensitive text, or special symbols. If you omit a range argument (e.g., 20 LAST or ALL), SED searches from the current line position onward.

To try it, type

```
* find "1850" ⤶
```

But SED displays the message,

```
String not found, correct the command:
```

Note the current line is line 6 and that the string we want is in line 1. SED searches from the current line to the end of the page, unless you include the argument ALL. So type

```
* find "1850"  all  ↵
```

```
1       In the 1850's both Charles Darwin and Alfred Russel Wallace
```

SED finds the string and highlights the line. To search a file of several pages (from the current position onward), use the argument IN PAGES REMAINING with the FIND command.

Setting Position and Inserting Page Breaks

By default, the current line position is the one where you pressed the ESC key. SED displays a range of lines around the current line, and the line itself is brighter than the others on your screen.

The POSITION command changes the current line or page position. And, as we mentioned earlier, many function keys allow you to change your position in the file quickly. To position the cursor on line 2, type

```
* po 2  ↵
```

```
2       proposed the theory of evolution.  Each theorist was basically
```

Function keys will allow you to move around in the file quickly. For example, the first function key in conjunction with the CTRL key positions the cursor on the first line on the page. Try it, press the FIRST LINE ON PAGE function key:

```
1       In the 1850's both Charles Darwin and Alfred Russel Wallace
```

And the second function key with the CTRL key brings you to the last line of the page. Press the LAST LINE ON PAGE function key:

```
6       surveyor's apprentice.
```

Now create two pages from one, using the SPLIT command. Postion the cursor on line 3; then type

```
* split ↵
```

SED breaks the file and moves lines 4 to 6 onto page 2.

```
split
*                                                        page 2 line 1
-------------------------------------------------------------------------
1       Darwin studied medicine in Edinburgh, and Wallace attended
2       the Working Men's Institute in Leicester, where he served as a
3       surveyor's apprentice.
```

To eliminate the second page, use the JOIN command.

```
* join ⤸                                                    page 1 line 6
-----------------------------------------------------------------------
```

SED always merges a page with the previous one. Use the POSITION command when joining pages, to position the cursor on line 1. The POSITION command is also useful when you want to insert text or move through a file.

Moving and Duplicating Text

The MOVE command relocates text from its current position to a different position in the file, or to another file. If the other file already exists, the MOVE command appends text to it. The command works with any group of lines that's less than a page.

Let's try out the MOVE command with HISTORY. The text looks like this:

```
1       In the 1850's both Charles Darwin and Alfred Russel Wallace
2       proposed the theory of evolution.  Each theorist was basically
3       a naturalist, although he did have a little background in science.
4       Darwin studied medicine in Edinburgh, and Wallace attended
5       the Working Men's Institute in Leicester, where he served as a
6       surveyor's apprentice.
```

To move line 2 before line 1, type

```
* move 2 before 1  ⤸

1       proposed the theory of evolution.  Each theorist was basically
2       In the 1850's both Charles Darwin and Alfred Russel Wallace
```

Now restore the correct order: repeat the line with CTRL-A:

```
* move 2 before 1 ⤸


1       In the 1850's both Charles Darwin and Alfred Russel Wallace
2       proposed the theory of evolution.  Each theorist was basically
```

Now take lines 1 through 3 out of HISTORY and move them onto another file, in this case, FILE1:

```
* move 1 3 onto file1  ⤸
FILE FILE1 CREATED
```

All that's left of the file is this:

```
1       Darwin studied medicine in Edinburgh, and Wallace attended
2       the Working Men's Institute in Leicester, where he served as a
3       surveyor's apprentice.
```

To restore the lines moved onto FILE1, position the cursor on line 1 and use the INSERT command to incorporate file FILE1 into HISTORY. (File FILE1 will remain on disk in the working directory; the INSERT command duplicates its contents into the SED file.) Type

```
* po 1 ↵
* insert from file1  ↵
```

HISTORY is now intact:

```
1       In the 1850's both Charles Darwin and Alfred Russel Wallace
2       proposed the theory of evolution.  Each theorist was basically
3       a naturalist, although he did have a little background in science.
4       Darwin studied medicine in Edinburgh, and Wallace attended
5       the Working Men's Institute in Leicester, where he served as a
6       surveyor's apprentice.
```

The DUPLICATE command works in much the same way as the MOVE command, but it doesn't remove the specified lines; it simply copies them to the destination address, which can be inside or outside the file. The destination argument in either the MOVE or DUPLICATE command can be BEFORE line-number, AFTER line-number, or ONTO pathname.

For example, duplicate the file by typing

```
* duplicate 1 6 after 6 ↵
```

And you see the source file twice. Delete the excess with the command

```
* del 1 6 ↵
```

The MOVE and DUPLICATE commands, in conjunction with the APPEND or INSERT command, can save you time when you need to create files that have many occurrences of similar text strings. You can simply copy the text in the file as needed.

Substituting Text

The SUBSTITUTE command allows you to replace one text string with another, in the text lines you specify. SED displays each changed line. If you omit a range argument, SED changes every matching text string from the current line to the end of the page.

For example, type

```
* sub "f" for "th" 1 2  ↵

1       In fe 1850's bof Charles Darwin and Alfred Russel Wallace
2       proposed fe feory of evolution.  Each feorist was basically
```

To correct the source file, reverse the arguments:

```
* sub "th" for "f" 1 2  ↵

1       In the 1850's both Charles Darwin and Althred Russel Wallace
2       proposed the theory oth evolution.  Each theorist was basically
```

And the text is corrected. Except, of course, for "Althred" and "oth," which shouldn't have been replaced by "th." These substitutions have to be corrected individually. Be careful of global replacements. They can do the unexpected!

Obtaining Help

As in the CLI, the SED text editor has a HELP command that provides information about SED commands and their formats. We didn't talk about the Help facility earlier because you need some background to understand its messages.

Entering the command HELP will produce a listing of Help topics. Type

```
* help ↵

                    ***** C O M M A N D S *****

ESCAPES      ADD TEXT   CHANGE TEXT     DELETE TEXT   LISTINGS   POSITIONINGS
-------      --------   -----------     -----------   --------   ------------
ABANDON      APPEND     MODIFY          DELETE        LIST       POSITION
BYE          INSERT     REPLACE         MOVE          VIEW       FIND
HELP         DUPLICATE  SUBSTITUTE      JOIN          PRINT
SAVE         UNDO       SPLIT
.
.
.
```

Or else, type the HELP command with the command you want information on:

```
* help append ↵

APPEND  [FROM <SOURCE> [<RANGE>]]

          ADDS TEXT TO THE END OF THE CURRENT TEXT....

       EXAMPLES: ....
```

THE HELP command is very handy when you're unsure of a command's syntax or capabilities.

Stopping SED

Whenever you want to end an editing session, type the BYE command. The command ends your editing session and updates your file by incorporating all changes made during the session. Then it returns control to the CLI. (If you want to continue in SED, but edit another file, press the BYE YES CONTINUE function key to close the current file, and open another by supplying a new filename.)

To end this session with SED, type

```
* bye ⤶

Output file - :UDD:ALEXIS:HISTORY
```

If you had changed an existing file, the SED program would have asked you about the original file:

Do you want to save the original file as a backup file?

If you don't want a backup file, type N or simply press NEW LINE. If you do want it saved, enter Y; SED will attach a .BU suffix to the original file, and display its name. For example,

```
Output file -  :UDD:ALEXIS:HISTORY
Backup file -  :UDD:ALEXIS:HISTORY.BU
```

The backup file is useful if you need to check the original contents of the file. If the backup file is a source program, you should use the CLI RENAME command to protect the version from deletion. SED only allows one backup file; so, next time you close an editing session, SED will delete the older backup file (filename.BU) and replace it with the newer backup file.

Command Files

Sometimes you'll want to make global edits of a file. Rather than issue the editing commands individually, SED allows you to execute a command file. The procedure is similar to a macro.

First you invoke SED; then build a small SED file of commands:

```
) xeq sed corrections ⤶

SED Rev. n   Input file - CORRECTIONS
Do you want CORRECTIONS to be created?    y ⤶

* append

1     duplicate 1 6 after 6 ⤶
2     duplicate 7 12 after 12 ⤶
3     delete 11 12 ⤶

(ESC)

* bye ⤶
```

Then execute SED with the command file, using the switch /PROFILE=command-file. For example, to make changes to HISTORY, type

```
) xeq sed/profile=corrections history  ⤶
```

SED displays its banner, and you'll see the commands flash before your eyes, as the editor implements each one. That's all there is to it. There's no need to spend hours entering repetitive commands.

Printing Files

You can print any text file with the CLI QPRINT command; for example, to print HISTORY, type

```
) qpr history   ↵
QUEUED, SEQ=924, QPRI=127
```

Now you can go to the line printer and pick up the printed file. (Chapter 2 explains how to operate the printer, if you need help.)

Summary

In this SED session, you've created and edited a file called HISTORY. It involved using the commands: APPEND, LIST, MODIFY, INSERT, DELETE, FIND, POSITION, SPLIT, JOIN, MOVE, DUPLICATE, SUBSTITUTE, and BYE. These commands will allow you to do nearly all the editing you want.

Four new text files result from the session: HISTORY, FILE1 (created by the SED MOVE command), the backup file HISTORY.BU, and the command file CORRECTIONS. SED also created a .SV and a .ED file for its own use.

What's Next?

If you're interested in learning another text editor, Chapter 5 explains SPEED, a character-oriented editor. Subsequent chapters demonstrate program development under AOS/VS and the Sort/Merge utility. Be sure to read Chapter 16, the documentation guide to AOS/VS products.

End of Chapter

Chapter 5
Writing with the SPEED Text Editor

The SPEED text editor, like the SED editor described in Chapter 4, is a utility program supplied with AOS/VS. The SPEED editor lets you create and modify files containing uppercase and lowercase ASCII text.

But SPEED is very different from SED because its commands are shorter and less mnemonic. SPEED has some powerful features: it can do arithmetic, execute commands conditionally and repetitively, and edit invisible characters and delimiters, such as NEW LINE. You can use either SED or SPEED. Both editors work with any text file. You'll find SED perfectly adequate for most of your editing needs. When you need to edit files for CTRL characters or other invisible characters, you'll find SPEED is a lifesaver.

As with the CLI and SED, the best way to learn SPEED is to use it. But, since SPEED is a little trickier than the CLI and SED, we give more preliminary information before starting the session.

To create and edit a file with SPEED, you must have Append and Write access — as you should always have for files within your user directory.

SPEED Features

SPEED is *character string oriented:* you insert and edit character sequences at the current character position. A character can be a space, a letter or number, or some other symbol.

To indicate the current character position, SPEED displays a *character pointer* (CP). On terminal screens, the CP appears as a blinking asterisk (*). Notice that SED, the line-oriented editor, emphasizes the current line position, while SPEED, the character-oriented editor, emphasizes the current character position. A line in SPEED refers to the characters between the CP and the next NEW LINE character. To insert or edit text, you place the CP where you want it, then begin to edit.

The SPEED Prompt and Delimiters

When SPEED is ready for a command, it displays an exclamation point (!) prompt. You type a command after the prompt and terminate the input by pressing CTRL-D. CTRL-D echoes as $$. For example, if you type the command 20L and press CTRL-D, it appears on your screen as 20L$$ with the dollar signs underlined.

In a change command, the ESC character serves as a text string delimiter. The ESC key also allows you to stack SPEED commands on a line before entering CTRL-D. For example, the command CHello(ESC)Bye (CTRL-D) echoes as CHello$Bye$$. We show the ESC key as $ in this chapter.

CTRL-D echoes as $$ but *it is not* the same as two dollar signs; the ESC key echoes as $ but is not the same as one dollar sign. The dollar sign (SHIFT-4) is just an ordinary text character, not a command delimiter.

If you enter multiple commands before entering CTRL-D, SPEED executes them sequentially, from left to right. If it finds an error, it describes the error, and ignores the remainder of the command line.

Edit Buffer

The edit buffer is an area of the computer's memory where SPEED works on your file during the editing process. The commands you type change the contents of the buffer; the buffer isn't written to disk until you type the FU or FB command.

The edit buffer holds the entire current page: what's marked by form feed characters (CTRL-L or 14_8); if there are no form feed characters in the file, the buffer holds the whole file.

SPEED Commands

Table 5-1 displays the SPEED commands that we will use in the session. It lists commands in the order we will use them. For a full description of SPEED commands, see the *SPEED Text Editor (AOS and AOS/VS) User's Manual.*

Table 5-1. Common SPEED Commands

Command	Action	Example
XEQ SPEED/*/D/* filename	Start the editor.	XEQ SPEED/D MYFILE ⤸
I	Insert text in the file.	Ihello CTRL-D
L	Move the CP to the beginning of a line.	20L CTRL-D
J ZJ	Jump to beginning or end of edit buffer.	J CTRL-D ZJ CTRL-D
S	Search for one or more characters.	Shello CTRL-D
C	Change one or more characters to other characters.	Chello$bye CTRL-D
K	Kill (delete) one or more lines of text.	2K CTRL-D
T	Type one or more lines of text.	10T CTRL-D
<commands ;>	Repeat (iterate) one or more commands.	<C ⤸ $ ⤸ ⤸ $;> CTRL-D
FU$H	File update and halt: include all edits to the file, write it to disk, and return to the CLI. The original file is deleted.	FU$H CTRL-D
FB$H	File backup and halt: same as FU, but also saves old file as a backup file.	FB$H CTRL-D

Control Keys

SPEED does not use as many screen edit/cursor control keys as the CLI and SED. Instead, different SPEED commands set position and make editing changes.

The CTRL sequences and special keys for SPEED are

Character	What it Does
CTRL-I or TAB key	Produces a tab.
CTRL-S	Suspends the screen display; it's useful for reading long files.
CTRL-Q	Continues the display suspended by CTRL-S.
CTRL-U	Deletes the line you are typing.
CTRL-D	Terminates and attempts to execute one or more SPEED commands. If you execute SPEED with the /D switch, you can press CTRL-D at any time to obtain a text display.
DEL	Deletes the preceding character.
CTRL-C CTRL-A	Interrupts the current SPEED command and returns the ! prompt.

Cursor and Case of Characters

When you are inserting text or typing a command, the cursor indicates the current position. The cursor is either a box superimposed on the current character, or an underscore that blinks beneath the current character.

SPEED commands are case insensitive. So, if your terminal has both uppercase and lowercase characters, you can use either case at will for commands. However, text is case sensitive. All language compilers accept either case in text strings, although FORTRAN 5 and COBOL compilers require *keywords* to be in uppercase. In the session, we use uppercase for SPEED commands and abbreviations; we use uppercase and lowercase for the text itself.

If You Make A Mistake

As you continue through the session, it's okay to make your own mistakes — everyone does. They help you learn the way SPEED works.

If, at any time, text seems to have vanished from the edit buffer, type the FB$H command and press CTRL-D to update the file and save the original version. Then examine both the current and original versions with either the SPEED editor or the CLI TYPE command. You can work with either one.

As with any program, the CTRL-C CTRL-B sequence will abort the SPEED process and return you to the CLI. However, the CTRL-C CTRL-B sequence nullifies all your edits as well, so use it only when you must. You can exit to the CLI with the FB$H command.

Invoking SPEED

To start the SPEED editor, use the CLI format line:

XEQ SPEED*[/D]* filename

The optional /D switch directs the SPEED editor to display a range of lines around the CP and to show the CP. On a display terminal, the /D switch makes SPEED *much* easier to use — because SPEED displays your position after each command. On a hard-copy terminal, omit the /D switch because printing the lines around the CP takes a lot of time.

In any case, if you omit the /D switch, you must use the T command to show the CP position or type the lines you want to see. In this session, we generally assume that you are working on a display terminal.

Once you enter the XEQ command line, SPEED will execute. If the file named in the command line already exists, SPEED will copy it into the edit buffer. If the file contains lowercase letters (as many text files do) SPEED will display the message *** Lower case input encountered *** just for your information.

If the file named in the command line does not exist, SPEED will query you for confirmation:

```
Create new file?
```

You will type Y and press NEW LINE to have SPEED create the file. Then you begin the edit session with an empty edit buffer.

The SPEED Session

To illustrate each SPEED command, we'll create and edit a source program named TOY.

To ensure that your user directory is the working directory; type

```
) dir/i  ⤸
```

And execute SPEED:

```
) x speed/d toy ⤸

SPEED REV n
Create new file?    Y ⤸
!
```

You received the *Create...* prompt because TOY did not exist before. You responded with Y to have SPEED create the file.

Inserting New Text (I)

To insert new text, use the I command. Each insert may include only one character or many lines of text.

Since SPEED is displaying its ! prompt, it's ready to accept a command. Let's insert text into the file using the I command. Type the following 47 characters *precisely as shown, but never including the* ! *prompt*. (CTRL-D counts as one character.)

```
! IThis is source line 1. ⤸
Thisis source line 2. ⤸
CTRL-D

This is source line 1.
Thisis source line 2.
*
!
```

Because we used the /D switch on the SPEED command line, SPEED displays the text and the CP. After each insertion, the CP points after the last inserted character. In this case the CP points to the character position after the ↓ that ends line 2. This lets you repeat insert commands in the same order that you would type words or lines of text on a typewriter.

Now let's try out SPEED's error checking. Try adding some text without the I command. Type

```
I This is source line 3. ↓
CTRL-D

* Confirm?
```

This question is typical of SPEED. Because we forgot to preface the new text with the I command, SPEED read our text as one of the commands that would take us back to the CLI, essentially losing all our edits in the process. But SPEED asked for confirmation first.

We don't want lose all our edits, so type

```
* Confirm?    N ↓
I
```

SPEED remains active because we answered N to the confirmation question. If ever SPEED prompts for confirmation, and you don't want to lose your edits, type N.

Let's include the I command and try it again. Type

```
I IThis is source line 3. ↓
CTRL-D

This is source line 1.
Thisis source line 2.
This is source line 3.
*
I
```

Forgetting the I command is a mistake that everyone makes. Correcting one early, you'll tend to remember it.

Because the I command is easy to forget, don't try to insert much text at once. To start, it's better to insert only a few lines, press CTRL-D, then review and correct what you've typed. Then continue inserting with a new I command.

Moving the CP to the Start of a Line (L)

The L command repositions the CP. For example, type

```
I -2L  CTRL-D

This is source line 1.
*Thisis source line 2.
This is source line 3.
I
```

As you can see, the L command moves the CP around. With a minus number, such as –2, it moves the CP backward 2 NEW LINE characters; with a plus number n it moves the CP forward n NEW LINE characters; and without any number, the L command moves to the beginning of the current line. If n is out of range, the L command simply moves the CP to the beginning of the first line or the beginning of the last line.

Type

```
! 1L CTRL-D
... (SPEED displays text and the CP) ...

! - 400L CTRL-D
... (SPEED displays text and the CP) ...

! 500L CTRL-D
... (SPEED displays text and the CP) ...

!
```

Sequential commands of 20L can help you read forward in longer files; commands of –20L help you read backward.

Jumping the CP to the Beginning or End of the Buffer (J, ZJ)

The J command repositions the CP to the beginning of the buffer; the ZJ command moves the CP to the end of the buffer. For example, type

`! J CTRL-D`
`* This is source line 1.`
...
	Go to the beginning of the buffer. SPEED displays all lines in the buffer, with the CP in position 1.

`! ZJ CTRL-D`
...
`This is source line 3.`
`*`
	Go to the end of the buffer.
	
	SPEED displays the last line, with the CP in the position following the NEW LINE delimiter.

`! IThis is line 4. J` `CTRL-D`	Append text to the buffer.

`This is source line 1.`
.
.
.
`This is line 4.`
`*`
`!`
	The exercise is complete.

As shown, the ZJ command is convenient when you want to append text to the file. You can simply start the insertion after the ZJ command executes. You'll find that the J command is easier than the –nL command when you want to position the cursor at the beginning of the buffer.

Searching for Characters (S)

SPEED searches from the CP onward. Type

`! Sline 1.CTRL-D`	Search for "line 1."
`Error:unsuccessful search`	The search failed.
`Sline 1.$`	
`!`	

In this case, the search failed because the CP was beyond the string we specified.

When a search command fails, SPEED puts the CP at the beginning of the buffer — so an identical search should succeed. Type

```
! Sline 1.CTRL-D              Search for "line 1."
This is source line 1.*       The search is successful.
.
.
.
!
```

When a search command succeeds, the CP is after the last character sought. To search for a NEW LINE, type

```
! S J                        Search for a "NEW LINE".
CTRL-D                       The search is successful.
*Thisis source line 2.
.
.
.
!
```

As you saw, SPEED can find unprinted (invisible) characters, such as a NEW LINE. You'll see a use for this later in this session.

The CP after any successful search is after the last character sought — in this example, after the NEW LINE on line 1, or the beginning of line 2.

For a search to succeed, you must type in precisely the string you want; for example, type in

```
! Sline2.CTRL-D               Search for "line2."
Error:unsuccessful search     The search is not successful because there is no space before 2.
.
.
.
!
```

SPEED looks for exactly what you type after the S command. For example, type

```
! S ThisCTRL-D                Search for " This".
Error:unsuccessful search     Search was not successful because a space prefaces the string.
SThis$
!
```

By default, SPEED does not distinguish case during searches. Type

```
! SLINE 1.CTRL-D              Search for "LINE 1."
This is source line 1.*       "Line 1." is found.
.
.
.
!
```

You'll be using the S command frequently, so try it a few more times:

```
! Shis CTRL-D          Search for "his ".
.
.
.
! Ss sCTRL-D          Search for "s s".
.
.
.
!
```

Changing a Character String (C)

The C command, like the S command, performs a search. But instead of simply searching, it searches and changes.

The same rules apply to the change command as to the search command: when a search fails, SPEED puts the CP at the beginning of the buffer; when a search succeeds, the CP is after the last character sought; SPEED looks for exactly what you type after the C command.

The C command has the form:

Cstring1$string2

where

string1	is the string you want to change.
string2	is the new string.
$	means to press the ESC key.

To delete a string, omit string2: e.g., Cstring1 CTRL-D.

Use the C command to correct the typographical error in line 2. Type

```
! J CTRL-D                Move to the beginning of the buffer.
! S1↓                     Position cursor at the start of line 2.
CTRL-D
*Thisis source line 2.    So far so good.
! Cis$ is CTRL-D          Change "is" to " is".
Th isis* source line 2.   This is not what you wanted.
.
.
.
!
```

With the C command, as with the S command, you need to be accurate and specific. Let's fix it by typing the following:

```
! J$CTh isis$This is CTRL-D    Combine the J and the C commands.
...
This is* source line 2.       This is where we wanted to be.
...
!
```

The C command can also extend changes over more than one line. For example, type

```
! Cline 4. ⤸                    Change "line 4." to itself, and add line 5.
$line 4. ⤸
This is source line 5. ⤸
CTRL-D
```

```
This is source line 1.          Done.
This is source line 2.
This is source line 3.
This is line 4.
This is source line 5.
*
!
```

The change command — like the search command — is useful. It allows you to edit in one step, rather than first positioning the CP with the search command, then inserting or deleting text.

Deleting Lines (K)

The C command allows you to delete characters and lines; but the K command is much more convenient. The command nK deletes n lines forward for a positive n, or n lines backward for a negative n.

To display the text, press

```
CTRL-D
```

```
This is source line 1.
This is source line 2.
This is source line 3.
This is line 4.
This is source line 5.
*
!
```

To delete (and restore) line 3, type

```
! J$S3$L CTRL-D                 Position the CP at the beginning of the right line.
* This is source line 3.        Done.
! 1K CTRL-D                     Delete it.
This is source line 2.          Line 3 is deleted.
* This is line 4.
! IThis is source line 3. ⤸     Insert line 3 again.
CTRL-D
This is source line 3.          Done.
```

To delete the remainder of a line — including the NEW LINE, simply put the CP before the doomed character string, and type 1K. For example, type

```
! S5$L$SThis CTRL-D                Position the CP before the string.
...
This* is source line 5.            The CP is positioned.
...
! 1K CTRL-D                        Delete the rest of the line.
...
This*                              Deletion complete.
! I is source line 5. ↓            Now restore the string.
CTRL-D
...
This is source line 5.             The string is reinstated.
*
!
```

Often it's easier to delete part or all of a bad line and insert a new one, than it is to try to correct the original. As you saw, the K command can help you do this.

With the K command, we suggest that you use only positive values of n to keep your editing simple. A minus number when the CP is in the middle of a line will delete not only the previous line, but also the left-hand portion of the current line.

At this point, you've executed the SPEED editor and created a file, inserted text, moved to different lines, jumped to the beginning and end of the buffer, searched for and changed text, and killed and recreated lines. You've used the commands XEQ, I, L, J, S, C, and K. These commands along with FU$H will suffice for most editing needs.

Typing lines (T)

The T command types one or more lines on the terminal. Because the /D switch on the SPEED command line instructs SPEED to type lines around the CP, and to show the CP, you rarely need to use the T command. But, the T command is useful when you want to see a different range of lines than the /D default, or lines far away from the CP, or even the entire buffer. If you didn't use the /D switch, you'd use the T command extensively to see what's going on.

When you use the T command, SPEED types only the specified number of lines. It shows the CP only if you omit an argument; e.g., if you enter `20T CTRL-D`, SPEED will not show the CP. The most common forms of the T command are as follows:

Format	Meaning
#T	Type the entire buffer.
T	Type the current line and show the CP.
nT	Type n lines, including the current line.
0,.T	Type the buffer from the beginning to the current position.
.,ZT	Type the buffer from the current position to the end.

For example, type

```
! #T CTRL-D
! 2T CTRL-D
```

Repeating (Iterating) Commands <commands;>

Sometimes you may want to execute one or more SPEED commands many times. To do this, use the form

< command *[command]* ... $; >

where command is any of those we've described, generally a command involving a search or change. Be sure to follow the last search or change command with a semicolon; this prevents the iteration command from looping. The semicolon *must go* after the last search or change command. If you forget the semicolon, you'll need to break the loop with CTRL-C CTRL-A.

For example, type

```
! J CTRL-D              Moves the CP to the beginning of the buffer.
...
! <C )                 Repeat <Change every NEW LINE
$ )                    to a NEW LINE
)                      and another NEW LINE.
$;> CTRL-D             Exit>, and do it.

*This is source line 1.

This is source line 2.

This is source line 3.

This is line 4.

This is source line 5.

!
```

Here, you changed every NEW LINE character in the buffer to two NEW LINE characters to produce double spacing. You could do the opposite to produce single spacing in the file. (Double spacing can be handy when you need to edit something.) This is one of SPEED's great features — you can edit NEW LINE and other invisible characters, such as CR and tabs.

File Update and Halt (FU$H); File Backup and Halt (FB$H)

The FU command writes the entire current edit buffer with your editing changes to disk, and then clears the buffer. The H command halts SPEED and returns control to the CLI. The FB$H command does the same thing but saves the original file as a backup, adding the characters .BU to the filename.

Type

```
! FU$H CTRL-D
)
```

In our practice session, there isn't an original file since we created TOY during this session.

Try re-editing TOY:

```
) x speed/d toy  J

This is source line 1.

This is source line 2.

This is source line 3.

This is line 4.

This is source line 5.

I CThis is$This is new CTRL-D

This is new* source line 1.

I FB$H CTRL-D

)
```

Because you requested a backup file this time, there will be both a TOY and a TOY.BU in the working directory — the latter being the backup file.

(Typing the full command line XEQ SPEED/D each time is a nuisance. Instead, you might want to ask your system manager to create a macro — named something like SPEED.CLI, in the :UTIL directory — that would allow you to type simply SPEED.)

Specifying a backup file can be useful if you need to check the original contents of the file. For a source program backup file, you should use the CLI RENAME command to attach a meaningful suffix to the backup file; for example, add .F77 to the filename.BU for a FORTRAN 77 source program.

SPEED allows only one backup file for any file; so, if a backup file exists, and you type the FB$H command, SPEED will delete the older backup file (without asking if you want to delete it) and replace it with the newer backup file.

Actually, you can type either the FU or FB commands without the H (meaning Halt), but since the commands clear the buffer, you'd have to type the H command anyway to get back to the CLI and re-execute SPEED. So you might as will type FU$H or FB$H. (There are commands to read the file back into the buffer from SPEED, but we don't describe them here.)

If you type the H command with a full buffer, SPEED will ask for confirmation (as it did during the session) before taking you back to the CLI. It does this so that an accidental H command will not nullify all your editing efforts. You might type H, and confirm with Y, if you hadn't changed the file. You might also exit with the H command if you realized that you had made substantial errors and wanted to start again.

Finally, we'll tell you that you can *omit* the ESC ($) delimiter after all but the Insert, Search, and Change commands. For example, the command line `FUH CTRL-D` is legal.

Summary

In this SPEED session, you've created and edited TOY, using the commands XEQ, L, J, S, C, K, <...>, FU$H, and FB$H. These commands will allow you to do nearly all the editing you want.

Two new files result from the session: TOY and TOY.BU; they are created by the SPEED FB$ command.

More Examples of SPEED Commands

Table 5-2 reviews each SPEED command we've used, and adds a more sophisticated example than what's shown in Table 5-1.

Table 5-2. SPEED Command Examples

Command	Example(s)	Meaning and Result
xeq speed	xeq speed/d file1 ↵	Execute SPEED with the /Display switch. If FILE1 exists, SPEED will read it into the edit buffer, otherwise SPEED offers to create it. The /D switch directs SPEED to display a range of lines after executing most commands.
I	IBAL=0. ↵ CTRL-D	Insert characters BAL=0. at the CP position.
J	JI* MYPROG ↵ CTRL-D	Jump to start of buffer, insert characters MYPROG there.
S	S(J.NE.0)CTRL-D	Search for the characters (J.NE.0).
C	CZZ9$ZZZ9CTRL-D	Search for characters ZZ9 and, if found, change them to ZZZ9.
L	L$I ↵ CTRL-D	Insert a NEW LINE character before the current line.
K	3K CTRL-D	Kill (delete) three lines.
T	#T CTRL-D	Type entire buffer.
< $;>	J$<CYes$No$;>CTRL-D	From start of buffer, change every NO or YES.
FUH and FBH	FUH CTRL-D	File update: write edited file to disk. Return to CLI.

What Next?

After the SPEED session, you have a working knowledge of the SPEED text editor. For more information, see the *SPEED Text Editor (AOS and AOS/VS) User's Manual*, described in Chapter 16. You're ready to proceed to a chapter about languages, or to the Sort/Merge utility, described in Chapter 15.

End of Chapter

Chapter 6
AOS/VS BASIC Programming

This chapter describes, in a hands-on session, how to develop an AOS/VS BASIC program. If you're totally unfamiliar with the AOS/VS BASIC language, we recommend that you turn to a language reference, listed in Chapter 16. Once you have a sense of the language, return to this chapter for an example of AOS/VS BASIC program development.

Program Development

This chapter leads you through a sample session in AOS/VS BASIC. Here are the steps you follow to create an AOS/VS BASIC program:

1. Enter the BASIC environment. In some systems, logging on brings you directly into BASIC, identified by the BASIC banner and * prompt; if so, continue to step 3. If you are logged on to the CLI, first set your searchlist. Type

   ```
   ) sea :util:BASIC [!sea] ↓
   ```

   If you receive an error message, try setting the search list to :BASIC.

2. Then bring up the BASIC environment from the CLI by typing

   ```
   ) xeq BASIC ↓
   ```

   (On some systems you may have a macro that sets your searchlist and executes the BASIC program in one step.)

3. Once in BASIC, write or edit a series of BASIC program statements. BASIC has its own advanced text editor and an interactive interpreter that rejects bad syntax as you type each statement.

4. Run the program with the BASIC command,

   ```
   * run ↓
   ```

5. If the program runs as you want it to, go to step 7.

6. Identify any problem using the BASIC runtime error messages or dynamic debugging. BASIC programs are easy to debug because you can insert breakpoints at will, print values at each breakpoint, and then continue the program. Having found the bug(s), go to step 3 and fix the offending statement(s).

7. Then store the program on disk with the command,

   ```
   * list "filename.BASIC" ↓
   ```

   Later, you can bring the program back into memory with the MERGE or NEW command.

8. If you have access to the CLI, you can choose to compile your BASIC program. The compiled version will run much faster than the interpreted one and won't require an interpreter for execution. If you choose not to compile, skip to step 14.

9. In order to compile, first store the program on disk with a BCI suffix, using the command,

   ```
   * save "filename.bci" ↓
   ```

10. Next, enter the CLI — either by calling it from BASIC with the CLI command, or exiting from BASIC with the command,

    ```
    * bye ↵
    ```

11. Invoke the BASIC compiler from the CLI with the command,

    ```
    ) bas/optimize filename ↵
    ```

12. Link the object modules into a program file from the CLi with the command,

    ```
    ) baslink filename ↵
    ```

13. Then run the compiled program from the CLI:

    ```
    ) xeq filename ↵
    ```

14. You're done! Either log off the system or return to the BASIC environment with the command:

    ```
    ) bye ↵
    ```

About AOS/VS BASIC

A BASIC program consists of a series of BASIC statements. Each AOS/VS BASIC statement begins with a number between 1 and 99999. BASIC checks the syntax of each line as you type it in. When you type the RUN command, the BASIC interpreter executes the statements sequentially by number. Thus, your program works.

AOS/VS BASIC variable and array names consist of 1 to 32 letters, numbers, or underscores; for example, VARIABLE_1. String variable names can be up to 32 letters, numbers, and underscores, and include a trailing $ or % — for example, FILE_1$. A string variable name that ends with $ indicates a varying-length variable, which is useful for filenames. A name that ends in % indicates a fixed-length variable, which BASIC will pad if necessary (a useful feature for fixed fields). All variable and string names must begin with a letter.

You can declare either an array name or string variable with the DIM (dimension) statement; for example, DIM ARRAY(100) or DIM STRING$*30. If you omit DIM, the default length of a string variable is 72. For comments, you can use either the REM (remark) statement or an exclamation point (!). With an exclamation point, you can comment each line directly, rather than including a separate line. For example,

```
100  rem  This is a comment.
110  !  This is also a comment.
120  dim  ARRAY(10,10)  ! dimension ARRAY.
```

While typing a program, you can use screen editing characters as with the SED text editor and CLI. You can abbreviate any keyword to its shortest unique string; for example, INP can represent INPUT.

You can examine program statements with the LIST command, and find keyword or variable names with the FIND command (FIND var or FIND "string"). To change statements individually use the EDIT command (EDIT line-number); to change globally use the the CHANGE command (CHANGE "string1" TO "string2" or CHANGE var TO var1).

When you're satisfied with a program, store it on disk with the LIST command (LIST "pathname"). Later, you can read it back into memory with the NEW command (NEW "pathname").

To start work on another program, type the NEW command, and continue to the next program. To sign off BASIC, type the BYE command.

You can have a BASIC program executed from the CLI by typing

```
) xeq BASIC filename ↵
```

where the filename is the name of a BASIC program. The BASIC environment also has its own CLI, which gives you many of the file managing abilities of the AOS/VS CLI.

To speed up the execution time of BASIC programs, you can compile the the AOS/VS BASIC source program. Once you write, run, and save an error-free program with the BASIC interpreter, return to the CLI to compile, link, and execute it. The BASIC compiler translates all BASIC statements into machine language instructions that almost always execute many times faster than the interpreted version.

Numeric Data Types

AOS/VS BASIC has two numeric data types: Real and Integer.

The default numeric data type is single precision real. as it is for many BASIC languages. A single precision real number has significance to about 6.7 digits. The other real type is double precision; its numbers have significance to about 14.6 digits.

Integer types are 16 bits long or 32 bits long.

You can select numeric types other than the default real with the DECLARE statement. The DECLARE statement can select a data type both globally and specifically. It can also dimension arrays, taking the place of a DIM statement. For example:

```
100  declare all REAL*8          ! Default is REAL*8.
110  declare integer I, J        ! I and J are integer.
120  declare real RARY(50)       ! RARY is dimensioned
                                 ! with 50 standard real
                                 ! elements.
```

HELP Commands

AOS/VS BASIC has both general and specific HELP commands to explain concepts and command syntax. As with the CLI, you type

```
* help ⤸
```

and BASIC displays a list of specific topics from which you can choose.

Invoking BASIC

If the BASIC banner appears as soon as you log on (type username and password), skip to the next section.

If the CLI prompt appears on your screen when you log on, you'll use a CLI command to enter the BASIC environment. Before you can use BASIC, the directory that holds the BASIC files, often directory :UTIL:BASIC or :BASIC, must be in your search list. (Sometimes there's a system macro that sets your searchlist for you and then executes BASIC.)

We suggest that you create a BASIC directory for your programs, and make it your working directory. This will put your BASIC programs in one place, and prevent conflicts with other programs that have the same names. For example, type

```
) dir/i ⤸
) cre/dir BASIC ⤸
) dir BASIC ⤸
```

With the last of the preceding commands, you are now in your CLI working directory BASIC. To enter the BASIC environment, type

```
) xeq BASIC  ↵
```

And BASIC will display its banner:

```
AOS/VS BASIC Revision  n  date  time
*
```

When the asterisk prompt appears, you are in BASIC and can start typing in commands and statements.

Practice Program

To familiarize yourself with AOS/VS BASIC, try this little program exercise. Some statements will be deliberately incorrect in order to show you AOS/VS BASIC error messages and recovery procedures. In BASIC, type

```
* 100  print  Test program"  ↵
Error n - Illegal statement or command syntax
```

The BASIC interpreter tests each statement as it's entered; in this case, there's a syntax problem. On your terminal screen, the BASIC interpreter highlights the place where it thinks the error began. The program name TEST PROGRAM should be enclosed in quotation marks. To insert the initial quotation mark, use the screen editing sequences. (We explain them in detail in Chapter 2 and summarize them at the beginning of Chapter 3.) To correct the line, enter the following screen editing sequences.

You enter:	**Action performed:**
CTRL-A	Press the CTRL key, hold it down, and type A. This displays last text typed: *100 print Test Program"*
HOME	Press the HOME key to move the cursor to the beginning of the line.
CTRL-F CTRL-F	Press CTRL-F twice to move the screen cursor to the T in *Test*.
CTRL-E	Press CTRL-E to open the line.
"	Type the missing quotation mark (").
CTRL-E	Press CTRL-E to close the line.
↵	Press the NEW LINE key to enter the line; this time BASIC accepts it because the syntax is correct.

Screen editing is very convenient; with practice you'll be able to fix bad lines in seconds.

Having fixed line 100, let's continue with the practice program. Type

```
* 110  let STRING$= "Result is" ↵
* 120  let a = 2  ↵
* 130  a1 = 3 ↵
* 140  a2  = 4 ↵
* 150  print STRING$; a + a1 / a2^ a     ! Get value. ↵
* 160  end  ↵

*  list  ↵
00100 print "Test program"
00110 let STRING$="Result is"
00120 let A=2
00130 let A1=3
00140 let A2=4
00150 print STRING$,A+A1/A2^A  ! Get value.
00160 end
```

As you can see, BASIC edits your input. It inserts leading zeroes as needed to produce 5 numbers on statements; it eliminates spaces in expressions; and it attempts to align comments (!) starting at column 50. It inserts implicit LET labels and expands keyword abbreviations. (It also makes keywords lowercase and identifiers uppercase, and indents DO and FOR/NEXT loops and drops unnecessary parentheses.)

```
* run  ↵

Test program
Result is  2.1875

END at 00160
```

The caret (^), produced by the SHIFT-6 keys, is the BASIC exponential operator. BASIC evaluated the expression in line 150 with 3 steps:

Step 1. A2^A = 16
Step 2. A1/16 = .1875
Step 3. A +.1875 = 2.1875

Now, store the program on disk, leave and re-enter BASIC, bring the program back into memory with the NEW command, run it, and leave BASIC again:

```
* list "Testprog.BASIC" ↵          Write it to disk.
* bye ↵                            Leave BASIC.
AOS/VS BASIC terminated ...

) xeq BASIC ↵                      Execute BASIC
     or                              or
username-password                  log on again.
 AOS/VS BASIC ...                  BASIC displays its banner.

* new "Testprog.BASIC" ↵           Bring program into memory.
* run ↵                            Run it.
Test program
Result is 2.1875

END at 0160
* bye ↵                            Leave BASIC.
```

Unless you store a program on disk, all work done on it during a BASIC session vanishes when you leave BASIC. In this case, the interpreter will ask for confirmation before it discards your program modifications.

You can use any valid system pathname to write a BASIC program to disk, as long as you have Append privileges to the directory or directories involved. To keep a program in the working directory, use its filename. If the filename you specify already exists, BASIC will display a message, asking if you want to delete the existing file. If you answer Yes, BASIC will overwrite it with the new one.

Writing a BASIC Sample Program

The program represented in the Figure 6-1 flow chart, and listed in Figure 6-2, is an AOS/VS BASIC variation of the C, COBOL, FORTRAN, and Pascal programs shown in other chapters. The program computes mortgage payments, taxes, and deductions in a general way, and displays its computations on the screen or sends them to the line printer.

The mortgage formulas are those used by U.S. banks, and the tax bracket formula is designed according to U.S. Internal Revenue Service regulations. Other mortgage and tax systems are used outside the U.S., so if you live elsewhere, treat the mortgage formulas and tax bracket as an example. (Later, you might want to replace the formulas with ones appropriate to your nation and individual situation.

To try the program, enter the BASIC environment. Use the NEW command to clear your work space, and then type in the MORTGAGE program, as shown in Figure 6-2. For example,

```
) xeq BASIC  ⤸
AOS/VS BASIC ...
*  new  ⤸
*  100  declare all REAL*8  !  To handle... ⤸
```

Before you begin to type in a new program, you should always type the NEW command to clear your storage area and prevent old program lines from mixing with new program lines. (This can be confusing at the very least.) You can also bring a program into memory with the command line "NEW program name", as long as the program named was created by the LIST — not the SAVE — command.

As you type in the program, you can examine the lines you've typed with the LIST command. To display a portion of the program, include the line numbers with the LIST command; for example, LIST 40, 70 or, LIST 100 TO 200. To change an existing line, type the EDIT command (EDIT line-number), and then use the screen editing characters to modify the line. To find a keyword or variable name, type the FIND command (FIND var or FIND "string"). To delete a line, type the DELETE command with the line number (DELETE line-number).

Periodically as you type, and again, when you're done, store your program by writing it to disk. Type the LIST command (LIST "MORTGAGE.BASIC", or whatever filename you prefer) to write the program to disk.

Even if you decide not to type the program, you should examine Figures 6-1 and 6-2 before continuing to the next section.

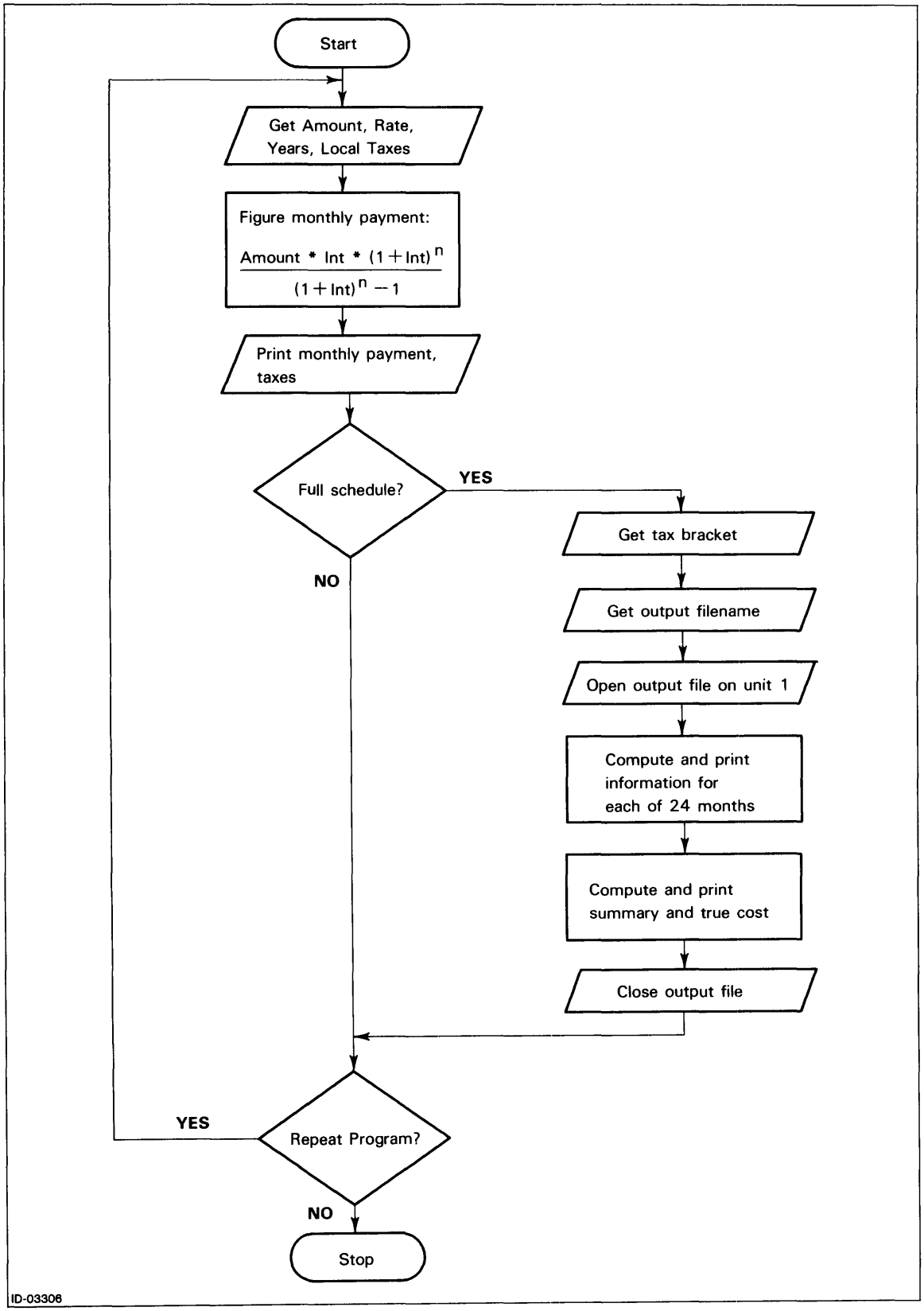


Figure 6-1. MORTGAGE Program Flow Chart (AOS/VS BASIC)

```
00100 declare all real*8                              ! To handle today's rates.
00110 declare integer YEARS,MONTHS,J
00120 print chr$(12)&" This program does mortgage payments, interest and taxes."
00130 do                                              ! Do until user exits ...
00140   print "Type amount, interest rate, loan life, and annual"
00150   print "property tax (0 if none). Separate entries with comma,"
00160   print "end with NEW LINE; e.g., 60000, 12.5, 25, 2000 NL"
00170   print
00180   print "   Amount? Rate? Years? Tax?"
00190   input prompt " ?   ":AMOUNT,RY,YEARS,TAX
00200   let RATE=RY/12/100                            ! Get monthly % RATE.
00210   let MONTHS=12*YEARS                           ! Get total number of MONTHS.
00220   rem  Compute monthly payment.
00230   let PAY=AMOUNT*RATE*(1+RATE)^MONTHS/((1+RATE)^MONTHS-1)
00240   rem  Define format F$ for dollar display. Repeats "-." 6 times.
00250   let F$=rpt$("-.",6)               ! RPT$ function, X6.
00260   rem  Print totals and give option for tax subroutine.
00270   print "Monthly payment:       Taxes:           Hideous total:"
00280   print using F$:PAY," ",TAX," ",PAY+TAX
00290   print
00300   print "Want a schedule of monthly costs for the first 2 years"
00310   print "and average monthly cost after US tax deductions?"
00320   input prompt "Type Y NL (Yes) for schedule, or NL (No)  ":Q$
00330   if (Q$="Y" or Q$="y") and SCHED then
00340   end if
00350   print
00360   input prompt "Type Y NL (Yes) to repeat program, NL to stop.  ":Q$
00370 loop while Q$="Y" or Q$="y"                 ! If not Y or y, stop.
00380
01000 def SCHED integer
01010   rem    Tax deduction/summary routine. Gets tax brkt and outfile name.
01020   print
01030   input prompt "What is your tax bracket as a fraction (e.g, .24)?  ":BRKT
01040   input prompt "Output file? Type L NL for printer, NL for console ":FILE$
01050   if FILE$(1:1)="L" then let FILE$="@LPT" else let FILE$="@CONSOLE"
01060   open #1:FILE$                                 ! Open on file number 1.
01070   rem    Set up variables and header. Then run the loop.
01080   let BAL=AMOUNT                                ! BALance decreases monthly.
01090   let ITD=0                                     ! Interest To Date starts w/0.
01100   output #1 using F$:"Amount=$",AMOUNT,"Rate=",RY,"%   Years=",YEARS
01110   output #1 using F$:"Payment=$",PAY,"  w/tax=$",PAY+TAX
01120   output #1:"      Month    Prin.     Int.    Int. total"        ! Header.
01130   for J=1 to 24                                 ! Loop for month = 1 to 24 ---
01140     let PRIN=BAL*RATE/((RATE+1)^MONTHS-1)       ! Get amnt of prin in pmnt.
01150     let ITD=ITD+(PAY-PRIN)                      ! Add interest to int total.
01160     let BAL=BAL-PRIN                            ! Decrement BALance.
01170     let MONTHS=MOTHS-1                          ! Decrement MONTHS.
01180     output #1 using F$:J,PRIN,PAY-PRIN,ITD      ! Write month's info.
01190   next J
01200   output #1                                     ! Blank line.
01210   let DSCRATCH=TAX*12+ITD/2                     ! Get annual deductions.
```

Figure 6-2. AOS/VS BASIC MORTGAGE Program with Errors (continues)

```
01220   let DEDUCTIONS=DSCRATCH-3400              ! Subtract 0 brkt amount.
01230   output #1:"Annual deductions are:      less $3400 std deduction is:"
01240   output #1 using F$:DSCRATCH," "," ",DEDUCTIONS
01250   output #1
01260   let COST=PAY+TAX-BRKT*DEDUCTIONS/12
01270   output #1:"                    ****SUMMARY****"
01280   output #1:"     Life:   Amount:    Rate(%):  Cash pay:   Brkt:   Cost:"
01290   output #1 using F$:YEARS,AMOUNT,RY,PAY+TAX,BRKT,COST    ! Write summary.
01300   close #1                                  ! Close open file.
01310   let SCHED=1                               ! Give the function a value.
01320 end def                                     ! Return to caller.
01330 end                                         ! End of the BASIC program.
```

Figure 6-2. AOS/VS BASIC MORTGAGE Program with Errors (concluded)

Running the BASIC Program

To run the BASIC MORTGAGE program that you just entered, or to run any program currently in memory, type the RUN command:

```
* run ⤸
```

The program executes, displays a program description, and prompts for the loan information:

```
This program computes mortgage payments, interest and taxes.
Type amount, interest rate, loan life, and annual
property tax (0 if none).  Separate each entry with a comma,
end with NEW LINE; e.g., 60000, 12.5, 25, 2000 NL

   Amount? Rate? Years? Tax?
?
```

Enter some plausible figures, separating each with a comma. For example, assuming a mortgage of $80,000 at 12.5% for 30 years, with local taxes of $980 per year, you would type

```
80000, 12.5, 30, 980 ⤸
```

and the program displays statistics on the loan:

```
Monthly Payment Taxes:          Hideous Total:

  853.81              980.00          1833.81
```

Then the program wants to know if you'd like a monthly schedule:

```
Want a schedule of monthly costs for the first 2 years
and average monthly cost after U.S. tax deductions?
Type Y NL (Yes) for schedule, or NL (No)
```

The total seems a little high — over $1,800 per month. Left out of the program was the conversion of the yearly tax figure to a monthly figure. It would be meaningless to continue, so don't request a schedule, but exit from the program by pressing the NEW LINE key. The system responds:

```
END at 01330
```

To fix the program error, insert a new line of code on line 215; type

```
* 215 TAX = TAX/12  ! Get monthly tax rate. ↵
```

and run the MORTGAGE program again:

```
* run  ↵
```

```
This program computes mortgage payments, interest and taxes.
Type amount, interest rate, loan life, and annual property tax ...
```

Enter figures as before, separating each entry with a comma:

```
      Amount? Rate? Years? Tax?
?     80000, 12.5, 30, 980 ↵
```

Then the program displays statistics on the loan:

```
Monthly Payment Taxes:          Hideous Total:

        853.81          81.67           935.47

Want a schedule of monthly costs for the first 2 years
and average monthly cost after U.S. tax deductions?
Type Y NL (Yes) for schedule, or NL (No)
```

These figures are more reasonable so we can continue with the program: Type

```
Y  ↵
```

```
What is your tax bracket as a fraction (e.g., .24)?
```

A tax bracket is the percentage of your income that's absorbed by government taxes. The last section of this chapter explains tax brackets in some detail. For now, use a figure of 24%, which is a plausible bracket:

```
.24 ⤶

Output file? Type L NL for printer, NL for console.
```

For this pass, let's view the mortgage report on the screen, so press

```
⤶

Amount=$ 80000.00    Rate= 12.50%      Years= 30.00
Payment = $  853.81    w/tax=$ 935.47
      Months    Prin.       ...

WARNING 71167 at 1170 invoked at 00330 - Identifier in expression has
not been assigned a value
Generated from MOTHS
1.00     20.47     833.33     833.33
WARNING 71167 at 1170 invoked at 00330 - Identifier in expression has
not been assigned a value
Generated from MOTHS
2.00      -80812.65      81666.45     ...     ...
WARNING 71167 at 1170 invoked at 00330 - Identifier in expression has
not been assigned a value.
Generated from MOTHS
.
.
.

Type Y NL (Yes) to repeat program, NL to stop.
```

Runtime errors! Let's stop the program and check line 1170, described in the first of many error messages. Press NEW LINE.

```
⤶
END at 01330
* list 1170 ⤶

01170 LET MONTHS=MOTHS-1 ! Decrement MONTH.
```

The problem is a typographical error, MOTHS instead of MONTHS. It occurred for each pass through the FOR-NEXT loop, eventually causing an overflow, hence the large number of messages. Warning reports about unused variables are a valuable feature of AOS/VS BASIC. Here, the message identified the problem immediately. Without the warnings, you would get invalid answers, and would have to work backward until you found the error.

To fix this error, use the EDIT command to display the line:

```
* edit  1170 ⤶
01170 LET MOTHS=MONTHS-1  ! Decrement ...
```

Then use screen editing characters to correct the typing error.

You enter:	Action performed:
CTRL-F CTRL-F	Moves the cursor to the first M.
→ (Do this 9 times.)	Rightarrow key brings the cursor to the second T.
CTRL-E	Opens the line for the text insert.
N	You type the missing N.
⤸	Enters the line.

The corrected line reads

```
01170 LET MONTHS=MONTHS-1      ! Decrement MONTHS.
```

Now run the MORTGAGE program, supplying the same figures (80000, 12.5, 30, and 980, with a tax bracket of .24):

```
* run  ⤸

This program computes mortgage payments, interest and taxes.
Type amount, interest rate, loan life, and annual property tax
... e.g., 60000, 12.5, 25, 2000 NL

   Amount? Rate? Years? Tax?
?    80000, 12.5, 30, 980 ⤸

Monthly Payment Taxes:          Hideous Total:
        853.81          81.67          935.47

Want a schedule of monthly costs for the first 2 years
and average monthly cost after U.S. tax deductions?
Type Y NL (Yes) for schedule, or NL (No)    Y ⤸

What is your tax bracket as a fraction?     .24 ⤸

Output file? Type L NL for printer, NL for console.       ⤸

Amount=$ 80000.00   Rate= 12.50%   Years= 30.00
Payment=$ 853.81     w/tax=$ 935.47
1.00    20.47    833.33         833.33
2.00    20.69    833.12         1666.45
3.00    20.90    832.90         2499.36
.
.
.

Annual deductions are:          less $3400 deduction is:
     10948.19                        7548.19
                ****SUMMARY****
Life:   Amount:  Rate(%):        Cash pay: Brkt:    Cost
30.00   80000.00   12.50         935.47   .24     784.51

Type Y NL (Yes) to repeat program, NL to stop.
```

Looks good — no runtime errors, and the figures seem reasonable. Try it once again; enter the information as before when prompted for it, but this time when you reach the following prompt, specify that output go to the line printer:

```
Output file?  Type L NL for printer, NL for console.    L ↵
```

This time no schedule of payments appears on the console because the program writes it to the printer. Then the program prompts

```
Type Y NL (Yes) to repeat program, NL to stop.
```

Because the program closed the output file, the output should appear immediately on the printer. Checking the printer, we find the 24 month payment schedule with tax computation, as shown in Figure 6-3.

Preceding the printed schedule is a header, listing your username, pathname to the temporary printer file, time, and date.

```
 Amount=$   80000.00  Rate=        12.50%   Years=     30.00
Payment=$     853.81  w/tax=$     935.47
      Month     Prin.      Int.    Int. total
       1.00     20.47    833.33     833.33
       2.00     20.69    833.12    1666.45
       3.00     20.90    832.90    2499.36
       4.00     21.12    832.69    3332.04
       5.00     21.34    832.47    4164.51
       6.00     21.56    832.24    4996.76
       7.00     21.79    832.02    5828.78
       8.00     22.01    831.79    6660.57
       9.00     22.24    831.56    7492.13
      10.00     22.47    831.33    8323.47
      11.00     22.71    831.10    9154.56
      12.00     22.94    830.86    9985.42
      13.00     23.18    830.62   10816.05
      14.00     23.43    830.38   11646.43
      15.00     23.67    830.14   12476.56
      16.00     23.92    829.89   13306.46
      17.00     24.17    829.64   14136.10
      18.00     24.42    829.39   14965.49
      19.00     24.67    829.14   15794.62
      20.00     24.93    828.88   16623.50
      21.00     25.19    828.62   17452.12
      22.00     25.45    828.36   18280.47
      23.00     25.72    828.09   19108.56
      24.00     25.98    827.82   19936.39

Annual deductions are:      less $3400 std deduction is:
  10948.19                        7548.19

                  ****SUMMARY****
     Life:   Amount:    Rate(%):  Cash pay:   Brkt:    Cost:
     30.00  80000.00     12.50     935.47       .24    784.51
```

Figure 6-3. Summary and Payment Schedule from MORTGAGE Program (AOS/VS BASIC)

You've corrected the program, so you can stop it and record it on disk using its original filename. BASIC prompts for verification before it writes over the original program. You provide it by typing Y:

```
END at 01330
* list "mortgage.BASIC" ↵
File already exists. Delete old copy?    Y ↵
```

To compile the BASIC program, continue to the next section. To leave BASIC and return to the CLI (or to log off, if you logged on BASIC directly), type

```
* bye ↵
```

Compiling the BASIC Program

Now, if you're interested in minimizing the execution time of your program, and have access to the CLI, you can compile the BASIC program. The compiler translates all BASIC statements in a program to create a file of machine language instructions.

In using the compiler, you create the program and debug it with the BASIC interpreter, as we just did. Then, before exiting from BASIC, save the program as a .BCI file. For example, to compile the MORTGAGE program, type

```
* save "mortgage.bci" ↵
```

Then exit to the CLI:

```
* bye ↵
```

Compile the program with the command

```
) bas/optimize mortgage ↵
```

Link it with the command

```
) baslink mortgage ↵
```

Finally, run the program under the CLI, noting the faster execution time:

```
) xeq mortgage ↵

This program computes mortgage payments, interest and taxes.
.
.
.
Type Y NL (Yes) to repeat program, NL to stop.
```

If your system has the SWAT debugger, you can use SWAT on the compiled version of the program. Include the /DEBUG switch on the compile and link command lines. Then execute the debugger with the command:

```
) swat filename ⤸
```

And that's all there is to it!

A Note on Itemized Deductions and Tax Brackets

The sample BASIC program assumes you are married and you itemize deductions. If you are single, change the 3400 in lines 1220 and 1230 to 2300 — e.g., `LET DEDUCTIONS= DSCRATCH -2300`.

It's easy to find your tax bracket if you don't know it. Look at the Tax Rate Schedules in the current U.S. Form 1040 instruction book. In Schedule X or Y, find the two dollar figures that bracket your net income, and moving one column to the right, find the figure followed by %. This, roughly, is your tax bracket because it indicates how much of the last taxable dollar went for taxes.

What Next?

If you want to experiment with another language, or review earlier material, turn to the appropriate chapter. Or, you can start writing your own BASIC programs, using the *AOS/VS BASIC Reference Manual.*

For details on AOS/VS BASIC manuals, and user documentation in general, turn to Chapter 16.

End of Chapter

Chapter 7
Business BASIC Programming

This chapter describes, in a hands-on session, how to develop a Business BASIC program under AOS/VS. If you're totally unfamiliar with Business BASIC, we recommend that you turn to a language reference, listed in Chapter 16. Once you have a sense of the language, return to this chapter for an example of AOS/VS program development.

Steps in Program Development

This chapter leads you through a sample session in Business BASIC. Here are the steps you follow to create a BASIC program:

1. Enter Business BASIC. Since the procedure varies from system to system, it's best to consult your system manager. Sometimes you'll log on directly into Business BASIC. Other times, you will execute the BASIC command from the AOS/VS CLI. Most often, you'll execute a macro, such as BBASIC.CLI from the AOS/VS CLI. For example, type the macro name,

   ```
   ) BBASIC ⤸
   ```

   The system displays the Business BASIC banner and the asterisk (*) prompt.

2. Write or edit a series of BASIC program statements. Business BASIC has its own editor and an interactive interpreter that rejects bad syntax as you type each statement.

3. Run the program with the BASIC command,

   ```
   * RUN ⤸
   ```

4. If the program runs as you want it to, go to step 6.

5. Identify the problem using BASIC runtime error messages or dynamic debugging. BASIC programs are easy to debug because you can insert STOP statements into the program, print values at each STOP statement, and then continue the program. Once the bug(s) are found, return to step 2 to correct them.

6. Store the program on disk with the command,

   ```
   * LIST "filename" ⤸
   ```

   Later you can bring it back into memory with the command,

   ```
   * ENTER "filename" ⤸
   ```

   where "filename" is the source program file.

7. You're done! Type

   ```
   * BYE ⤸
   ```

   to log off the system or return to the AOS/VS CLI.

About Business BASIC

A Business BASIC program is a series of BASIC statements, each beginning with a number between 1 and 9999. BASIC checks the syntax of each line as you type it in, and displays an error message if anything is wrong. When you type the RUN command, Business BASIC executes the statements sequentially by number. Thus, your program works.

Business BASIC is an integer BASIC, it doesn't support floating point numbers. Its variable and array names must begin with a letter, and can be followed by up to five alphanumeric characters. You can declare either a string variable or array name with the DIM (dimension) statement; e.g., DIM R$(30) or DIM B(100), where the lower boundary of the array is 0. Thus an array dimensioned as (10,10) has 121 places not 100. Business BASIC doesn't allow default dimensioning. For comments, you use the REM (remark) statement.

While you are typing in the program, or at any point, you can examine its statements with the LIST command. You can change a statement by typing its line number, then the new text. When you're satisfied with a program, write it to disk with the LIST command. Later, you can read it back into memory with the ENTER command. From BASIC, you can print the program on the line printer by typing the LIST command with "@LPT" as an argument.

To start work on another program, type the NEW command and begin typing the program. You can execute a BASIC program only from BASIC; you can't do it from the AOS/VS CLI.

Invoking BASIC

If the Business BASIC banner and prompt (*) appears on your terminal when you log on, then skip to the next section.

If the AOS/VS CLI appears on your terminal screen when you log on, you'll use the CLI to enter Business BASIC. Before you can use BASIC, the directories that hold the BASIC files and the library files, often directories :UTIL:BBASIC and :UTIL:BBASIC:$SYSLIB, must be on your search list. Secure this information from your system manager, and set the search list with the command:

```
) SEARCHLIST :UTIL:BBASIC :UTIL:BBASIC:$SYSLIB [!SEA]  ⤸
```

The new search list remains in effect until you log off or modify it.

Before you start programming in Business BASIC, we suggest that you create a Business BASIC directory and make it your working directory. This will put your programs in one place and prevent conflicts with other programs that have the same names. For example, type

```
) DIR/I  ⤸
) CRE/DIR BBASIC  ⤸
) DIR BBASIC  ⤸
```

You are now in the working directory BBASIC. To enter Business BASIC, use the CLI XEQ command or your system macro; for example, type

```
) XEQ BBASIC ⤸
```

Business BASIC will display its banner:

```
   Welcome to AOS/VS Business BASIC Rev n - DOUBLE PRECISION
.
.
.
*
```

When the asterisk prompt appears, you are in Business BASIC and can start typing commands and statements.

Practice Program

To familiarize yourself with AOS/VS Business BASIC, try typing this little program:

```
*  5  DIM S$(10)  ⤸
* 10  PRINT  "TEST PROGRAM" ⤸
* 20  LET S$=  "RESULT IS" ⤸
* 30  LET A = 2 ⤸
* 40  LET A1 = 3 ⤸
* 50  A2 = 4 ⤸
* 60  PRINT S$;  A + A1 + A2^A ⤸
```

Now display the program:

```
* LIST ⤸

0005  DIM S$[10]
0010 PRINT "TEST PROGRAM"
0020 LET S$="RESULT IS"
0030 LET A=2
0040 LET A1=3
0050 LET A2=4
0060 PRINT S$,A+A1+A2^A
```

The next step is to execute the program:

```
* RUN ⤸

TEST PROGRAM
RESULT IS  21
*
```

The caret (^), produced by the SHIFT-6 keys, is the BASIC exponential operator. BASIC evaluated the expression in line 60 with:

Step 1. A2^A = 16

Step 2. A1+16 = 19

Step 3. A+19 = 21

Now, record the program on disk, leave BASIC, re-enter BASIC, enter the program, and leave BASIC again:

`* LIST "TESTPROG" ⤸`	Write it to disk.
`* BYE ⤸`	Leave BASIC.
`) XEQ BBASIC ⤸`	Execute Business BASIC
or	or
Username-Password	log on again.
Welcome to AOS/VS Business ...	Business BASIC banner.
`* ENTER "TESTPROG" ⤸`	Bring program into memory.
`* BYE ⤸`	Leave Business BASIC.

Unless you store a program on disk, all work done on a program during a BASIC session vanishes when you leave Business BASIC.

You can use any valid system pathname to record (LIST) a BASIC program on disk, as long as you have Append privileges to the directories involved. (Generally, it's best to store the file in the working directory.)

Writing the Business BASIC Example Program

The BASIC program represented in the flow chart in Figure 7-1 and listed in Figure 7-2 is a variation of the AOS/VS BASIC, FORTRAN and COBOL programs shown in other language chapters. The program computes monthly mortgage payments, based on the term of the loan, the principal, and the interest rate. It also displays its computations on the terminal screen and/or sends them to the line printer.

Since Business BASIC is an integer-based language, the program demonstrates ways to perform floating point calculations and print results using decimal point notation. In this way, we are able to show one MORTGAGE program as it's developed in Business BASIC, AOS/VS BASIC, C, COBOL, Interactive COBOL, FORTRAN 77 and Pascal.

The mortgage formulas are those used by U.S. banks. Different formulas are used outside the United States. If you live outside the United States, treat the program's formula as an example. (Later, you might want to replace it with an appropriate formula.)

We recommend that you type the program from Business BASIC. (If you have access to the AOS/VS CLI, you can use a text editor to type in the Business BASIC progams rather than the BASIC editor. But then you must access Business BASIC and enter the program. Business BASIC will reject all erroneous lines at once, which can be confusing. Later on, when you know the system better, you may choose to use the SED or SPEED text editor to type in your Business BASIC programs. Then you bring the program listing into memory with the command, `ENTER "listing-file"`.)

So, to try the program, enter Business BASIC, as described above. Type the NEW command, and then type in the MORTGAGE program, as shown in Figure 7-2. The program has some deliberate errors that will test the error handling of AOS/VS and the Business BASIC interpreter.

```
*  NEW ⤸
*  10 DIM BAL$[8],Q$[1],PAY$[8],NUMER$[8],POWER$[8],FILE$[10]  ⤸
.
.
.
*  70 END  ⤸
```

The NEW command clears your storage area. This prevents old program lines from intermixing with new program lines, which can be confusing at the very least.

As you type in the program, you can examine the lines you've typed with the LIST command. To display specific lines, include the range as an argument. For example, to display lines 100 through 200, you would type

```
* LIST 100, 200 ↵
```

After you type a block of text and, again, when the program is all typed, use the LIST command to write the program to disk; for example, type

```
* LIST "MORTGAGE" ↵
```

You can also receive a hardcopy of what you've written of the program by typing the LIST command with "@LPT" as an argument.

Even if you decide not to type the program, you should examine Figures 7-1 and 7-2 before continuing to the next section.

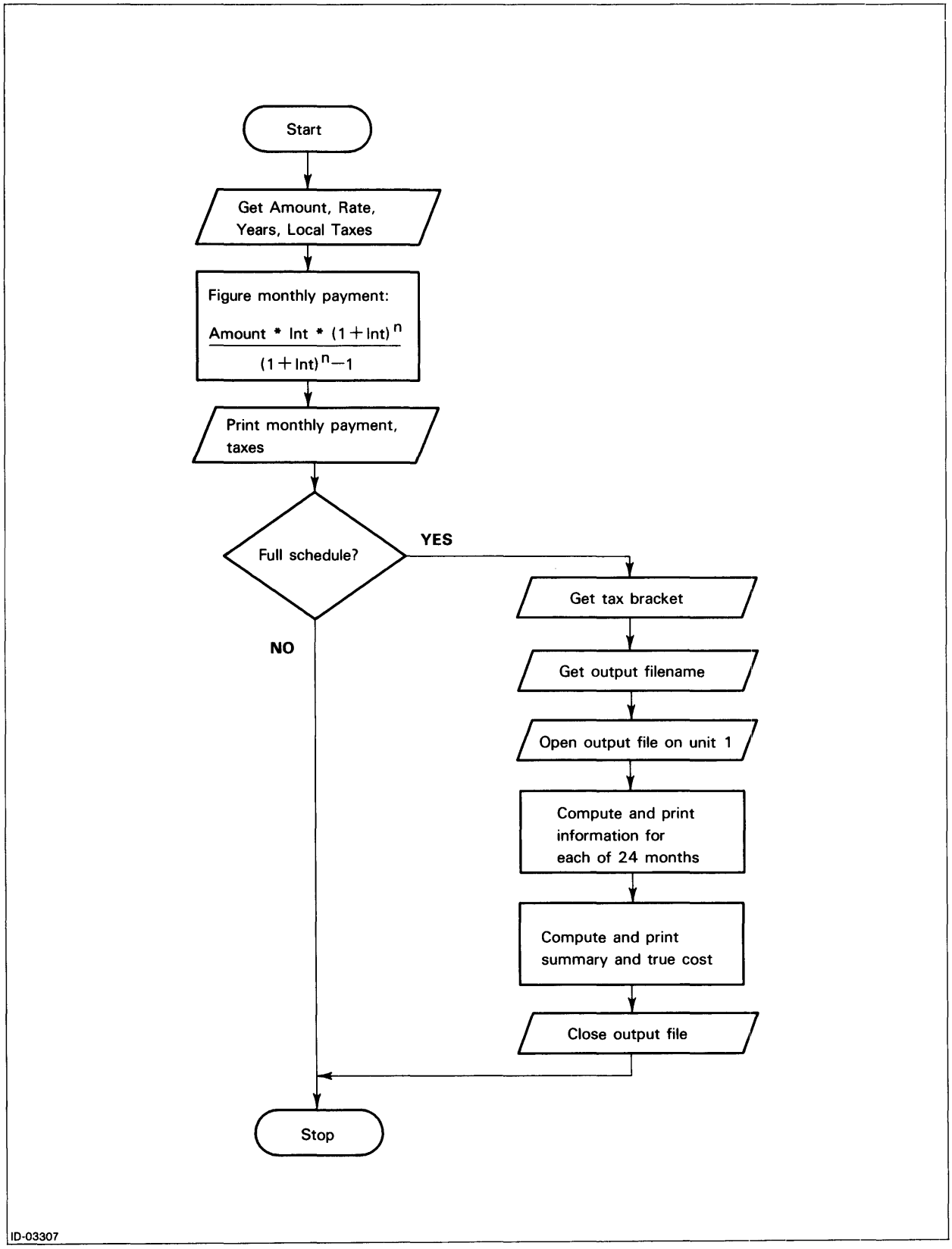


Figure 7-1. MORTGAGE Program Flow Chart (Business BASIC)

```
0010 DIM BAL$[8],Q$[1],PAY$[8],NUMER$[8],POWER$[8],FILE$[10]
0020 LET PAY,PRIN,RESULT,REMAIN,AMSHFT=0
0030 PRINT @(-30);@(-20);" This program calculates monthly mortgage "
0040 PRINT "payments once you enter the amount, interest rate, "
0040 PRINT "term of the loan, and annual property tax.  It then prints "
0050 PRINT "a two year summary of payments. "
0060 PRINT
0070 PRINT "Enter the interest rate as an integer, whose last two"
0075 PRINT "digits represent decimal values; for example, "
0080 PRINT "enter 14.5% as 1450, 12.75% as 1275."
0100 PRINT
0110 INPUT "Enter amount of the loan, in dollars:  $ ",AMOUNT
0120 INPUT "Enter interest rate (express 12.5% as 1250):  % ",RY
0130 INPUT "Enter loan life in years:  ",YEARS
0140 INPUT "Enter the property tax per year:   ",YRTAX
0150 LET TAX=(YRTAX*100)/12
0160 REM   Shift the amount of the loan left to save accuracy when
0170 REM   dividing later in the program.
0180 IF AMOUNT>=100000000 THEN GOTO 0230
0190 LET AMOUNT=AMOUNT*10
0200 LET AMSHFT=AMSHFT+1
0210 GOTO 0180
0220 REM   Calculate length of loan in months.
0230 LET MONTHS=12*YEARS
0240 DIM POWER[MONTHS]
0250 REM   Calculate montly interest rate.
0260 REM   The yearly rate was entered as a four digit number, with two
0270 REM   decimal places.  Shift the yearly rate left to preserve
0275 REM   accuracy before dividing.
0280 LET RATE=(RY*1000)/12
0290 REM   Compute Montly Payments
0300 REM   Rate is now less than one and accurate to 7 digits,
0310 REM   so to calculate (1+rate)^months, 1 is adjusted to be 10^7.
0320 LET POWER[1]=10000000+RATE
0330 REM   Calculate exponential:  (1+rate)^months
0340 REM   This portion of the program calculates the powers of (1+rate).
0350 REM   This direct method is used to preserve accuracy in the
0360 REM   calculations and prevent an overflow error.  Store the
0370 REM   powers of (1+rate), since the program needs them later.
0380 FOR J=1 TO MONTHS-1
0390   QMUL POWER$,POWER[J],POWER[1]
0400   QDIV POWER[J+1],POWER$,10000000,REMAIN
0410   IF REMAIN>=450000 THEN LET POWER[J+1]=POWER[J+1]+1
0420 NEXT K
0430 REM   Calculate (1+rate)^months - 1
0440 LET DENOM=POWER[MONTHS]-10000000
0450 REM   Using quadratic precision, calculate amount*(1+rate)^months
0460 QMUL NUMER$,POWER[MONTHS],AMOUNT
0470 REM   Divide this result by (1+rate)^months - 1
0480 QDIV RESULT,NUMER$,DENOM
```

Figure 7-2. Business BASIC MORTGAGE Program With Errors (continues)

```
0490 REM   Get monthly payment by multiplying the above by monthly rate.
0500 QMUL PAY$,RESULT,RATE
0510 REM   Truncate unnecessary precision and store the result in PAY.
0520 QDIV PAY,PAY$,100000
0530 REM   Round monthly payment off to two decimal places by truncating
0540 REM   the number of digits shifted earlier and rounding up.
0550 IF MOD(PAY,10^AMSHFT)>=5*10^(AMSHFT-1) THEN LET PAY=PAY+10^AMSHFT
0560 LET PAY=PAY/10^AMSHFT
0570 REM
0580 REM    Print totals and give option for tax subroutine.
0590 PRINT
0600 PRINT "    Monthly payment:    Taxes:          Total:"
0610 PRINT USING "T11,2(D10.2),T37,D10.2",PAY,TAX,PAY+TAX
0620 PRINT
0630 PRINT "Would you like a schedule of monthly costs for the first 2 years"
0640 INPUT "and the average monthly cost after U.S. tax deductions? (y/n)",Q$
0650 IF Q$="y" OR Q$="Y" THEN GOSUB 0710
0690 STOP
0700 REM   Tax deduction/summary subroutine.  Gets tax bracket and outfile name.
0710 PRINT
0720 INPUT "What is your tax bracket as a percentile (e.g., 24)",BRACK
0730 INPUT "Output file?   Type L NL for printer, NL for console  ",FILE$
0740 IF FILE$="1" OR FILE$="L" THEN LET FILE$="@LPA" ELSE LET FILE$="@CONSOLE"
0750 CLOSE
0760 OPEN FILE[0,1],FILE$
0770 REM   Make the current balance equal to the intial amount.
0780 LET BAL=AMOUNT/(10^(AMSHFT-2))
0790 REM   ITD is interest to date.
0800 LET ITD=0
0810 PRINT FILE[0]
0820 PRINT FILE[0]
0830 PRINT FILE[0],USING "'Amount=$',D10.2,' Rate=',D5.2,'%  Years=',D2.0",AMOUNT/10^(AMSHFT-2),RY,YEARS
0840 PRINT FILE[0],USING "'Payment=$',D10.2,'  with tax=$',D10.2",PAY,PAY+TAX
0850 PRINT FILE[0]
0860 PRINT FILE[0],"     Month     Prin.        Int.   Int.total"
0870 FOR I=1 TO 24
0880   REM   Use values for (1+rate)^months found earlier.
0890   REM   Create a quad precision balance to preserve accuracy in the
0900   REM   next divide.
0910   QMUL BAL$,BAL*(10^(AMSHFT-2)),RATE
0920   REM   Calculate principle as bal*rate/((1+rate)^months - 1)
0930   QDIV PRIN,BAL$,POWER[MONTHS]-10000000
0940   REM   Round off to two decimal places.
0950   IF AMSHFT>4 THEN LET PRIN=PRIN/10^(AMSHFT-4)
0960   IF MOD(PRIN,100)>=50 THEN LET PRIN=PRIN+100
0970   LET PRIN=PRIN/100
0980   REM   Update interest to date and balance
0990   LET ITD=ITD+(PAY-PRIN)
1000   LET BAL=BAL-PRIN
1010   LET MONTHS=MONTHS-1
1020   PRINT FILE[0],USING "T7,D2.0,3(D12.2)",I,PRIN,PAY-PRIN,ITD
1030 NEXT I
1040 PRINT FILE[0]
```

Figure 7-2. Business BASIC MORTGAGE Program With Errors (continued)

```
1050 REM   Calculate tax deductions and cost
1060 LET DSCRAT=YRTAX*100+ITD/2
1070 LET DEDUCT=DSCRAT-340000
1080 PRINT FILE[0],"Annual deductions are:       less $3400 std deduction is:"
1090 PRINT FILE[0],USING "T10,D10.2,T45,D10.2",DSCRAT,DEDUCT
1100 PRINT FILE[0]
1110 LET COST=PAY+TAX-(BRACK*DEDUCT/1200)
1120 PRINT FILE[0], TAB(32);"****SUMMARY****"
1130 PRINT FILE[0]," Life:       Amount:      Rate(%):     Cash pay:      Bracket:         Cost:"
1140 PRINT FILE[0],USING "'      ',D2.0,5('     ',D10.2)",YEARS,AMOUNT/10^(AMSHFT-2),RY,PAY+TAX,BRACK,COST
1150 CLOSE
1160 RETURN
1170 END
```

Figure 7-2. Business BASIC MORTGAGE Program With Errors (concluded)

Running the BASIC Program

To run the Business BASIC program that is currently in memory, type the RUN command:

```
* RUN ⤸
```

The MORTGAGE program executes, displays an initial message, and prompts for the mortgage statistics:

```
This program calculates monthly mortgage payments once you enter
the amount, interest rate, term of the loan, and annual property
tax.  It then prints a two year summary of payments.
```

Since the program uses decimal notation to compute the mortgage payments, you enter the interest rate as an integer. For example, enter 12.5% as 1250 and 13.25 as 1325.

Now the program prompts for mortgage figures, and we enter some plausible figures:

```
Enter amount of the loan, in dollars:     $60000 ⤸
Enter the interest rate (write 12.5% as 1250):     %1325 ⤸
Enter loan life in years:         20 ⤸
Enter property tax per year:      1125 ⤸

Error 22 at 0420 - NEXT - no FOR
```

Something is wrong — let's look at that part of the program. Type

```
* LIST 380, 420  ⤸

0380 FOR J=1 TO MONTHS-1
0390    QMUL POWER$,POWER[J],POWER[1]
0400    QDIV POWER[J+1],POWER$,10000000,REMAIN
0410    IF REMAIN>=450000 THEN LET POWER[J+1]=POWER[J+1]+1
0420 NEXT K
```

Looking at the program, you can easily see that line 420 is wrong: K should be J. To correct the problem, type

```
* 420 NEXT J ⤸
```

Now, try the program, entering the same figures as before. Type

```
* RUN  ⤸

.
.
.
Enter amount of the loan, in dollars:     $60000 ⤸
Enter the interest rate (write 12.5% as 1250):     %1325 ⤸
Enter loan life in years:         20 ⤸
Enter property tax per year:      1125 ⤸
```

This time, the program runs smoothly and displays statistics on the loan:

```
Monthly Payment:      Taxes:      Total:
         713.65       93.75       807.40
```

Then it prompts for instructions. Answer Yes.

```
Would you like a schedule of monthly costs for the first 2 years and the
average monthly cost after U.S. tax deductions? (y/n)     Y ↵
```

The mortgage program now asks you for your tax bracket. (Because interest deductions can lower your tax liability, the true cost of a mortgage must take into account the tax dollars saved. In this sample mortgage program, we assume you are a married person, itemizing only mortgage-related expenses. The current U.S. Internal Revenue Service publications describe all these procedures.)

```
What is your tax bracket as a percentile?
```

To calculate your tax bracket, see the IRS Tax Rate Schedules, or estimate the percentage of your Gross Adjustable Income that is taxable. A common percentage is 25%; so, if you're uncertain about your particular tax bracket, type

```
25 ↵
```

Finally, the program wants to know where to send the output. Let's try the terminal screen the first time:

```
Output file?  Type L NL for line printer, NL for console.    ↵
```

Since we selected a screen display, the following appears:

```
Amount=$  60000.00 Rate=13.25%  Years=20
Payment=$    713.65  with tax=$    807.40

     Month      Prin.        Int.    Int.total
       1        51.16      662.49       662.49
       2        51.72      661.93      1324.42
       .          .          .           .
       .          .          .           .
       .          .          .           .
      24        65.58      647.79     15730.48
```

So far, so good. Now run the program a second time, requesting a line printer listing, rather than a screen display:

```
* RUN  ⤸

.
.
.
Enter amount of the loan, in dollars:     $60000 ⤸
Enter the interest rate (write 12.5% as 1250):     %1325 ⤸
Enter loan life in years:        20 ⤸
Enter property tax per year:     1125 ⤸

Type L [NL] for line printer listing, [NL] for console.     L ⤸

I/O ERROR n AT 760 - File write protected
*
```

A runtime error! Check the offending line:

```
* LIST  740, 760 ⤸

0740 IF FILE$="1" OR FILE$="L" THEN LET FILE$="@LPA" ELSE LET FILE$="@CONSOLE"
0750 CLOSE
0760 OPEN FILE[0,1],FILE$
```

The problem, which the program flagged at line 760, actually occurs on line 740 where the program tried to write to a printer device (@LPA) instead of the printer queue (@LPT). The system needs the queue to help manage multiuser printer requests; so it requires users to open the queue, not the device.

To fix the line, first close all files with the CLOSE command (always a good idea if a program bombs during file input or output):

```
* CLOSE ⤸
```

Then type a new line 740 that correctly refers to the printer's queue name:

```
* 740 IF FILE$="1" OR FILE$="L" THEN LET FILE$="@LPT" ELSE LET FILE$="@CONSOLE" ⤸
```

Run it again, giving the same figures.

This time there is no runtime error, but there is a slight delay as the program writes the schedule to the printer. Then it stops.

Because the program closed the printer file, the output should appear immediately at the printer. You should find there a 24-month payment schedule with tax computation. This schedule is shown in Figure 7-3.

A header precedes the printed schedule. The header has our username, pathname to the temporary printer file, time, and date.

```
Amount=$  60000.00 Rate=13.25%  Years=20
Payment=$    713.65  with tax=$    807.40

     Month      Prin.        Int.   Int.total
       1        51.16      662.49      662.49
       2        51.72      661.93     1324.42
       3        52.29      661.36     1985.78
       4        52.87      660.78     2646.56
       5        53.46      660.19     3306.75
       6        54.05      659.60     3966.35
       7        54.64      659.01     4625.36
       8        55.25      658.40     5283.76
       9        55.86      657.79     5941.55
      10        56.47      657.18     6598.73
      11        57.10      656.55     7255.28
      12        57.73      655.92     7911.20
      13        58.36      655.29     8566.49
      14        59.01      654.64     9221.13
      15        59.66      653.99     9875.12
      16        60.32      653.33    10528.45
      17        60.99      652.66    11181.11
      18        61.66      651.99    11833.10
      19        62.34      651.31    12484.41
      20        63.03      650.62    13135.03
      21        63.72      649.93    13784.96
      22        64.43      649.22    14434.18
      23        65.14      648.51    15082.69
      24        65.86      647.79    15730.48

Annual deductions are:       less $3400 std deduction is:
            8990.24                          5590.24

                              ****SUMMARY****
  Life:        Amount:       Rate(%):      Cash pay:       Bracket:         Cost:
    20       60000.00          13.25         807.40           0.25         690.94
```

Figure 7-3. Payment Schedule from the Mortgage Program (Business BASIC)

You've fixed the program, so you can stop it and write it to disk. First you must delete the old version with the command,

```
* DELETE "MORTGAGE" ↵
```

And then write the new version to disk using its original name:

```
* LIST "MORTGAGE"  ↵
```

You can also print the program from BASIC by typing

```
* LIST "@LPT"  ⤵
```

To leave Business BASIC and return to the CLI (or to log off, if you logged on into BASIC), type

```
* BYE  ⤵
```

What Next?

If you want to experiment with another language, continue to the appropriate chapter. Or, perhaps you want to review earlier material or begin writing your own programs. Chapter 16 contains a bibliography of Business BASIC manuals, and many other products available under AOS/VS.

End of Chapter

Chapter 8
C Programming

This chapter describes, in a hands-on session, how to develop and run a C program under AOS/VS. If you're totally unfamiliar with the C language, we recommend that you turn to a C language reference, listed in Chapter 16. Once you have a sense of the language, return to this chapter for an example of AOS/VS program development.

Program Development under AOS/VS

These are the steps you follow to develop a program in AOS/VS C:

1. Create or edit a C source file with the SED or SPEED text editor:

   ```
   ) xeq sed filename.c ↵
   or
   ) xeq speed/d filename.c ↵
   ```

   With your chosen editor, type in the C statements and comments that make up the program.

2. Compile the source file with the compile macro supplied by Data General:

   ```
   ) cc/l=filename filename ↵
   ```

   Instead of sending output to a disk file, you can send it to a line printer with the /L=@LPT switch. The compile line can also include the /DEBUG, /LINEID, or /N switch, all described in the chapter.

3. If there are any error messages from the compiler, return to step 1 and fix the incorrect statement(s). If there are no error messages, go to step 4.

4. Link the object file to produce an executable program:

   ```
   ) ccl filename ↵
   ```

   This command can include the /DEBUG switch, described later.

5. Execute the program with the CLI command line,

   ```
   ) xeq filename ↵
   ```

6. If the program runs as you want it to, go to step 9.

7. Identify logic errors using runtime error messages or erroneous output. Or, if you included the /DEBUG switch on the compile and link commands, and your system has the SWAT™ interactive debugger, debug the program with the command:

   ```
   ) swat filename ↵
   ```

8. Go to step 1 and fix the erroneous statement(s).

9. You're done! You can go on to write your own C programs under AOS/VS.

This chapter guides you through all the steps you need to develop and execute a C program.

The C Program Example

The sample program is a simple program to calculate home mortgage payments; it produces a schedule of monthly principal and interest. The program uses only ordinary arithmetic operations, calls no subroutines, and is less than two pages long. The program uses two formulas. You don't need to understand how these formulas work to understand the illustration.

The mortgage formulas are those used by U.S. banks. Different formulas are used outside the United States. If you live outside the United States, treat the formula parts of the program as examples only. (Later, you might want to replace these formulas with one's appropriate to your nation.)

Figure 8-1 shows a flow chart for this program. Figure 8-2 shows the initial version of the program MORTGAGE.C itself, complete with errors.

Even if you decide not to duplicate this program, you should, when you're reading the text, examine Figures 8-1 and 8-2 before continuing to the next section.

To compile, link, and run C programs, your system must have the appropriate compiler files and libraries. The directory that holds these — often directory :UTIL:C — must be in your search list. Set your search list with the command

```
) sea :util:c [!sea] ↓
```

The C directory will remain on your search list until you either modify it or log off the system.

Writing the C Program

If you want to try the MORTGAGE program, we suggest that you create a directory for it and use that directory as your working directory. This will prevent conflicts with other programs that have the same names and will encourage you to place all your C programs in the same place. For example, type

```
) dir/i  ↓
) cre/dir c  ↓
) dir c  ↓
```

Now, use the SED or SPEED text editor to create the source file. We suggest that the source filename end with the conventional C suffix, .C.

```
) xeq sed mortgage.c  ↓
```

or

```
) xeq speed/d mortgage.c  ↓
```

Type in the program according to Figure 8-2. Notice that the program is written in a block format, each block enclosed in braces ({ }). Although C doesn't require any special format, we urge you to use this style for clarity. The C language uses semicolons to set off statements, and identifies comments with paired slashes and asterisks: /* opens a comment; */ closes it. C is primarily a lowercase language; you'll see in Figure 8-2 that uppercase is the exception rather than the rule. Names can include up to 32 of the following characters: A–Z, a–z, 1–9, _, $, and @. (Note that the @ symbol has a unique meaning in the UNIX™ operating system and the SWAT debugger, so it's best to avoid it.)

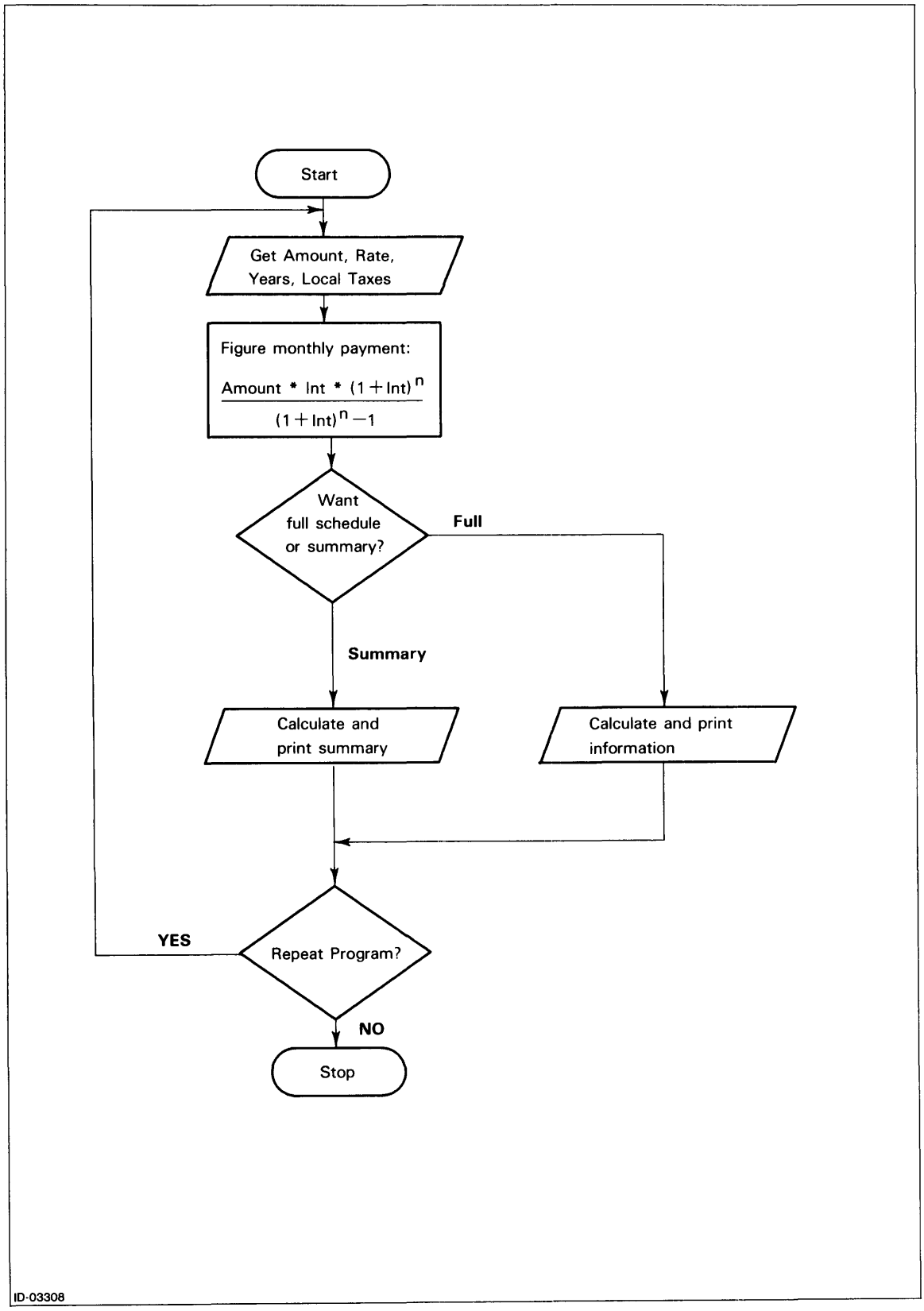


Figure 8-1. MORTGAGE Program Flow Chart (C)

```
/* MORTGAGE.C computes mortgage payments with a summary or full schedule.  */

#include <ctype.h>
#include <stdio.h>
#include <math.h>
#define LSIZE   256             /* Input line size */
double  payment();              /* Calculate load payment */
void    summary();              /* Print out summary */
void    full();                 /* Print out full schedule */

/*  Main function to calculate the mortgage.  */

int main(){
        double  amount;         /* Principal amount */
        double  rate;           /* Interest rate */
        double  pay;            /* Amount to pay */
        int     years;          /* Number of years to pay */
        char    line[LSIZE];
        char    ans;
                /* Loop calculating the mortgage */
        do {
                /* Request the principal, interest rate and number of years */
                fprintf( stderr, "\nEnter principal amount:    $ " );
                fgets( line, LSIZE, stdin );
                sscanf( line, "%f", &amount );
                fprintf( stderr, "Enter interest rate:       %% " );
                fgets( line, LSIZE, stdin );
                sscanf( line, "%f", &rate );
                rate /= 100.0;          /* Convert from percentage */
                fprintf( stderr, "Enter the number of years:   " );
                fgets( line, LSIZE, stdin );
                sscanf( line, "%d", &years );

                        /* Calculate the monthly payment */
                pay = payment( amount, rate, years );
                        /* See what the user wants */
                fprintf( stderr, "Type f or F for a full schedule." );
                fprintf( stderr, "  Type s or S for a summary. " );
                fgets( line, LSIZE, stdin );
                sscanf( line, " %c", &ans );
                switch( ans ){
                        case 's':
                        case 'S':
                                summary( amount, rate, years, pay );
                                break;
                        case 'f':
                        case 'F':
                                full( amount, rate, years, pay );
                                break;
                        default:
                                fprintf( stderr, "Unknown input\n" );
                                break;
                }
```

Figure 8-2. The C Mortgage Program with Errors (continues)

```
                        /* Finally, see if the user wants to do this again */
                fprintf( stderr, "Do you want to repeat this program?" );
                fprintf( stderr, "  Enter Yes or No:  " );
                fgets( line, LSIZE, stdin );
        } while( line[0] == 'Y' || line[0] == 'y' );
        return 0;       /* indicate no errors */
} /* Main */

/*  Payment function -- calculate and print the monthly payment.  */

double payment( amount, rate, years )
double amount;                          /* Principal amount */
double rate;                            /* Interest rate */
int    years;                           /* Number of years to pay */
{
        double  r;                      /* Monthly rate */
        double  pay;                    /* Monthly payment */
        int     months;                 /* Number of months to pay */
        r = rate / 12.0;
        months = years * 12;
                /* Calculate monthly payment */
        pay = amount * r * pow((1.0 + r), (double)months ) /
              (pow( (1.0 + r), (double)months ) - 1.0);
        return( pay );
} /* Payment */

/*  Summary -- print a summary of the rates.  */

void summary( amount, rate, years, pay )
double amount;                          /* Principal amount */
double rate;                            /* Interest rate */
int    years;                           /* Number of years to pay */
double pay;                             /* Total payment */
{
        printf( "\n" );
        printf( "Amount          = %$9.2d\n", amount );
        printf( "Interest Rate   = %11.4f\n", rate );
        printf( "Loan life       = %6d years\n", years );
        printf( "Monthly Payment = %$9.2f\n\n" pay );
        return;
} /* Summary */
/*  Full function -- print out the full payment schedule.  */

void full( amount, rate, years, pay )
double  amount;                         /* Loan amount */
double  rate;                           /* Interest rate */
int     years;                          /* Years to pay */
double  pay;                            /* Monthly payment */
{
        double  balance;                /* Balance to pay */
        double  interest;               /* Interest to pay */
        double  int_to_date;            /* Interest pay to date */
```

Figure 8-2. The C Mortgage Program with Errors (continued)

```
        double  principal;              /* Principal remaining */
        double  r;                      /* Monthly rate */
        int     months;                 /* Number of months to pay */
        int     orig_months;            /* Original number of months */
        int     i;                      /* Loop index */
                /* Initialize variables */
        r           = rate / 12.0;
        orig_months = months = years * 12;
        balance     = amount;
        int_to_date = 0.0;

                /* Write out monthly payment to the standard output */
        printf( "\n" );
        printf( "Amount          = %$9.2f\n", amount );
        printf( "Interest Rate   = %11.4f\n", rate );
        printf( "Loan life       = %6d years\n", years );
        printf( "Monthly Payment = %$9.2f\n\n", pay );
        printf( " Num        Interest      Prin. Pay      Prin. Bal." );
        printf( "    Interest Paid to Date\n" );

                /* Calculate each month's interest */
        for( i = 1; i <= orig_months; i++ ){
                principal    = balance * r / pow( (r+1), (double)(--months) );
                balance     -= principal;
                interest     = pay - principal;
                int_to_date += interest;
                printf( "%3d %$16.2f %$14.2f %$14.2f %$17.2f\n",
                        i, interest, principal, balance, int_to_date );
        }
        return;
} /* Full */
```

Figure 8-2. The C MORTGAGE Program with Errors (concluded)

Compiling the C Program

Having created an initial version of MORTGAGE.C, you now compile it. Since the initial version of a program usually has some syntactical errors, use the /N switch on the compile command so the compiler won't create an object module. Type

```
) cc/n mortgage.c  ↵
```

The compiler displays its banner, with the date, time, revision number, and the switch option you selected. Then it displays several error messages:

```
Error 502 severity 2 beginning on line 1024 (line 92 of file mortgage.c)
        printf( "Monthly Payment = %$9.2f\n\n" pay  );
                                               ^

Syntax Error.
A symbol of type "binary_op" has been inserted before this symbol.
("+" was used.)

Error 373 severity 3 beginning on line 1024 (Line 92 of file mortgage.c)
        printf( "Monthly Payment = %$9.2f\n\n" pay  );
                                               ^
.
.
.
```

C error messages are very specific. These identify line 92 as having an error in syntax error. A caret identifies the position where the error occurs. On a second look you can see there should be a comma after the second quotation mark.

Using the SED or SPEED text editor, correct the error, producing the following line:

```
        printf( "Monthly Payment = %$9.2f\n\n",  pay  );
```

After leaving the editor, recompile the program with the /LINEID and the /DEBUG switches:

```
) cc/lineid/debug mortgage.c  ↵
```

The /LINEID switch directs the C compiler to report the program line numbers involved if any runtime errors occur. This switch can help you identify logic errors in your sources; it is useful during program development whenever a program might have runtime errors. The /DEBUG switch directs the compiler to build a symbol table, so you can use SWAT or some other debugger to track runtime problems.

The compiler completes without error messages, and we receive the CLI's prompt. Now we can link the program with the system Link utility.

Creating the Program File with Link

You link the MORTGAGE program with the command:

```
) ccl/debug mortgage.ob ↵
```

Before Link builds the MORTGAGE program file, it displays its program banner and the switches selected.

```
LINK REVISION n   ON  date  AT  time
OPTIONS LINK/UNX/DEBUG
MORTGAGE.PR CREATED
```

The C Link macro, CCL includes the /UNX switch, directing the utility to include libraries necessary for the interface between the C language and the UNIX operating system. The command line includes the /DEBUG switch, allowing later use of the SWAT debugger.

In C, or any other high-level language, Link errors are rare. If you ever do receive a Link error message, and the text doesn't provide advice about fixing the problem, see the *AOS/VS Link and Library File Editor (LFE) User's Manual.*

Executing the C Program

The next step is to execute MORTGAGE.PR. To execute this or any other C program, simply type the XEQ command and follow it with the program name and press NEW LINE.

```
) xeq mortgage ↵
```

The mortgage program prompts for loan figures: you should supply some plausible figures, like those that follow:

```
Enter principal amount:     $ 57500 ↵
Enter interest rate:        % 13.5 ↵
Enter number of years:      30 ↵
```

Then the program queries you for the type of payment schedule you want to see. For the purpose of this example, reply with S.

```
Type f or F for a full schedule.  Type s or S for a summary.   s ↵
```

Since you requested a summary, the MORTGAGE program displays a brief report, with the terms of the mortgage and the monthly payment due:

```
Amount          =          $0
Interest Paid   =        0.1350
Loan Life       =       30 years
Monthly payment =     $658.61
```

But there's something wrong here because you entered a principal of $57,500 and the summary cites an amount of $0. Let's try the program again, this time providing different figures. Since the program offers the option to repeat, we simply request another run:

```
Do you want to repeat this program?  Enter Yes or No:      Yes ↵
```

Try again:

```
Enter principal amount:      $  80000 ↵
Enter interest rate:         %  12.5 ↵
Enter number of years:             30 ↵

Type f or F for a full schedule.  Type s or S for a summary.     s ↵
```

The program displays the summary:

```
Amount          =          $0
Interest Paid   =        0.1250
Loan Life       =       30 years
Monthly payment =     $853.80
```

The same problem exists: the value displayed in the *Amount* line is still 0, not the figure we supplied. Let's stop the program and run MORTGAGE.PR in the SWAT debugger to detect the problem. To stop the program, press CTRL-C CTRL-A:

```
 ^C ^A

*ABORT*
Signal SIGNIT (interrupt) aborted program
ERROR: From Program
xeq, mortgage
```

On Using the SWAT® Debugger

Data General's SWAT® Debugger, if your system has it, can really ease the debugging phase of program development. To use SWAT, you compile your program with the /DEBUG switch, as you did in the previous section "Compiling the C Program." Now you use it again and also ask for a listing with the /L=@LPT switch:

```
) cc/debug/l=@lpt mortgage.c  ↵
```

After compiling the program, link it using the /DEBUG switch on the link macro:

```
) ccl/debug mortgage.ob  ↵
```

Then start up the program in SWAT:

```
) swat mortgage   ↵
AOS/VS SWAT Rev n ...
PROGRAM -- :UDD:ALEXIS:C:MORTGAGE
>
```

In SWAT, you can set breakpoints by listing a line number with the BREAKPOINT command, or can check logic by displaying source lines with the LIST command. To start or run the program, enter the CONTINUE command; to leave the program, exit with the BYE command. The TYPE command allows you to examine variables at breakpoints, whereas the DESCRIBE command provides information on program symbols: their type, storage class, location or size.

A SWAT session with MORTGAGE can uncover the reason the program is displaying an erroneous amount in the mortgage summary. First, let's use the DESCRIBE command to verify that the program symbols in the MORTGAGE program are correctly defined. The symbols are Amount, Rate, and Years. Type

```
> describe amount ↵
amount (4 words at +6R8 words) auto double
> describe rate ↵
rate (4 words at +12R8 words) auto double
> describe years ↵
years (2 words at +22R8) auto int
```

The mortgage program defines Amount and Rate as double precision floating point numbers, with a length of 64 bits; it defines Years as an integer of 2 words, with a length of 32 bits. The symbols are defined correctly, so the problem isn't one of definition.

The next step is to trace the variable Amount through the program, and see if it's correctly labeled throughout. We can use the LIST command to display the file. Type

```
> list main ↵
1061 int main( ){
> list 1061 10000 ↵
int main(){
        double  amount;         /* Principal amount */
        double  rate;           /* Interest rate */
        double  pay;            /* Amount to pay */
.
.
.
```

Amount appears many times in the listing; three lines are of special interest:

```
1062    double  amount;         /* Principal amount */
1073            sscanf( line, "%f", &amount );
1137    printf( "Amount          = %$9.2d\n", amount );
```

Line 1062 above defines Amount as a double precision floating point number. In line 1073, Amount is labeled type "f", a floating point number. But then, line 1137 directs the program to print Amount as type "d" — as an integer. Here's the problem! The program used a floating point number as an integer.

Because the variable Amount is labeled incorrectly, the program responded with a nonsensical answer. It's a simple problem to correct. Exit from the debugger, and run the SED or SPEED text editor to correct the line.

```
> bye ↵
This SWAT session is terminating.

) xeq sed mortgage.c ↵
```

or

```
) xeq speed/d mortgage.c ↵
```

Change the line to read:

```
        printf( "Amount          = %$9.2f\n", amount );
```

Now recompile, link, and run the program. You can combine the commands to do so in a single line:

```
) cc/link/debug mortgage.c; xeq mortgage ↵
```

The compiler and Link programs display messages; then the MORTGAGE program runs and prompts you for loan figures:

```
Enter principal amount:     $  80000 ↵
Enter interest rate:        %  12.5 ↵
Enter number of years:           30 ↵
```

Next, select a summary to see if the problem is corrected:

```
Type f or F for a full schedule.  Type s or S for a summary.     s ↵

Amount          =   $80000
Interest Paid   =     0.1250
Loan Life       =    30 years
Monthly payment =   $853.80
```

The amount displayed is correct, and we find a monthly payment due of $853.80. The program runs accurately.

Now look at the full schedule of mortgage payments. However, instead of displaying the schedule on the screen, direct it to a disk file, so you can later print it. To redirect the output, you need to execute the program again from the CLI. So exit from the program:

```
Do you want to repeat the program?  Type Yes or No:     no ↵
```

You use the /L=filename switch on a program-name argument to send output to a disk file; if the filename already exists, use the /O=filename switch, which deletes the existing file and writes output to the newly created one. You can use the /L=@LPT switch, in much the same way, to send output to a printer.

To send output to a file called MORTGAGE.OUT, type

```
) xeq mortgage/l=mortgage.out  ↵
```

The program prompts for Amount, Rate, Term, and the type of schedule. Reply to them as before, and request a full schedule. However, this time you won't see anything on the screen. Instead, after a while, the program prompts to repeat:

```
Do you want to repeat the program?  Type Yes or No:     no )
```

The program terminates, and you return to the CLI, where you can display the file MORTGAGE.OUT with the CLI TYPE command. Or, to produce a hard-copy version of the file, use the QPRINT command:

```
) qpr mortgage.out  )
QUEUED, \ SEQ=n
```

Checking the printer, you'll find the full payment schedule, part of which is shown in Figure 8-3. The entire schedule is eight or so pages long. The interest total at the end (which the Figure 8-3 doesn't show but your listing will) is pretty appalling.

Preceding the full schedule is a printed header that describes our username, output file pathname, and date.

```
Amount          = $80000.00
Interest Rate   =      0.1250
Loan life       =     30 years
Monthly Payment =   $853.80

 Num        Interest      Prin. Pay      Prin. Bal.    Interest Paid to Date
  1          $833.61         $20.19       $79979.80           $833.61
  2          $833.41         $20.39       $79959.41          $1667.02
  3          $833.20         $20.60       $79938.81          $2500.23
  4          $832.99         $20.81       $79918.00          $3333.22
  5          $832.78         $21.02       $79896.97          $4166.00
  6          $832.56         $21.23       $79875.74          $4998.57
  7          $832.35         $21.45       $79854.28          $5830.93
  8          $832.13         $21.66       $79832.61          $6663.06
  9          $831.91         $21.88       $79810.73          $7494.98
 10          $831.69         $22.11       $79788.61          $8326.68
  .             .               .              .                .
  .             .               .              .                .
  .             .               .              .                .
 42          $823.32         $30.48       $78949.80          $34809.66
```

Figure 8-3. Beginning of the Full Payment Schedule from the MORTGAGE Program (C)

What Next?

If you want to experiment with another language, or review earlier material, continue to the appropriate chapter. You can also start writing your own C programs, using the *C Language Reference and Runtime Manual (AOS/VS).*

For more details about C or SWAT manuals — and AOS/VS user documentation in general — see Chapter 16.

End of Chapter

Chapter 9
COBOL Programming

This chapter shows you how to develop and run a COBOL program under AOS/VS. If you're totally unfamiliar with COBOL, we recommend that you turn to a language reference listed in Chapter 16. Once you have a sense of the language, return to this chapter for an example of AOS/VS program development.

Steps in Program Development under AOS/VS

These are the steps you follow to develop a COBOL program under AOS/VS:

1. Write and, if necessary, edit a COBOL program source file with a text editor such as SED or SPEED:

   ```
   ) XEQ SED filename.COB ↓
   ```

   or

   ```
   ) XEQ SPEED/D filename.COB ↓
   ```

2. Compile the source file with the compile macro supplied by Data General:

   ```
   ) COBOL/L=@LPT filename ↓
   ```

3. If there are any error messages from the compiler, go to step 1 and fix the errors with your chosen editor.

4. Build the object file into an executable program file with the Link macro supplied by Data General:

   ```
   ) CLINK filename ↓
   ```

5. Execute the program with the CLI command,

   ```
   ) XEQ filename ↓
   ```

6. If the program runs the way you want it to, skip to step 9.

7. Identify logic errors using runtime error messages or erroneous output. If your system has the SWAT debugger, you can use it to debug programs. Use the /DEBUG switch on both the compile and link macros, and then execute SWAT with the command,

   ```
   ) SWAT filename ↓
   ```

8. Go to step 1 and fix the incorrect code.

9. You're done! You can go on to write your own COBOL programs under AOS/VS.

The COBOL Program Example

The program in this chapter, like the program in other language chapters, calculates home mortgage payments. It can also produce a month-by-month schedule of principal and interest. Figure 9-1 is a flow chart of the program, and Figure 9-2 is the COBOL program itself, with errors deliberately written into it.

The mortgage formulas are those used by U.S. banks. Different formulas are used outside the United States, so if you live outside the United States, treat the formula parts of the program as examples only. (Later, you might want to replace the existing formulas with ones appropriate to your nation.)

To compile and run COBOL programs, your system must have the appropriate compiler files and libraries. The directory that holds these, often directory :UTIL:COBOL, must be in your search list. To set the search list, you would type

```
) SEA :UTIL:COBOL [!SEARCHLIST]↵
```

The COBOL directory will remain on your search list until you reset it, or log off the system.

Even if you decide not to copy the sample COBOL program, you should — in addition to reading the text — examine Figures 9-1 and 9-2 before writing original programs.

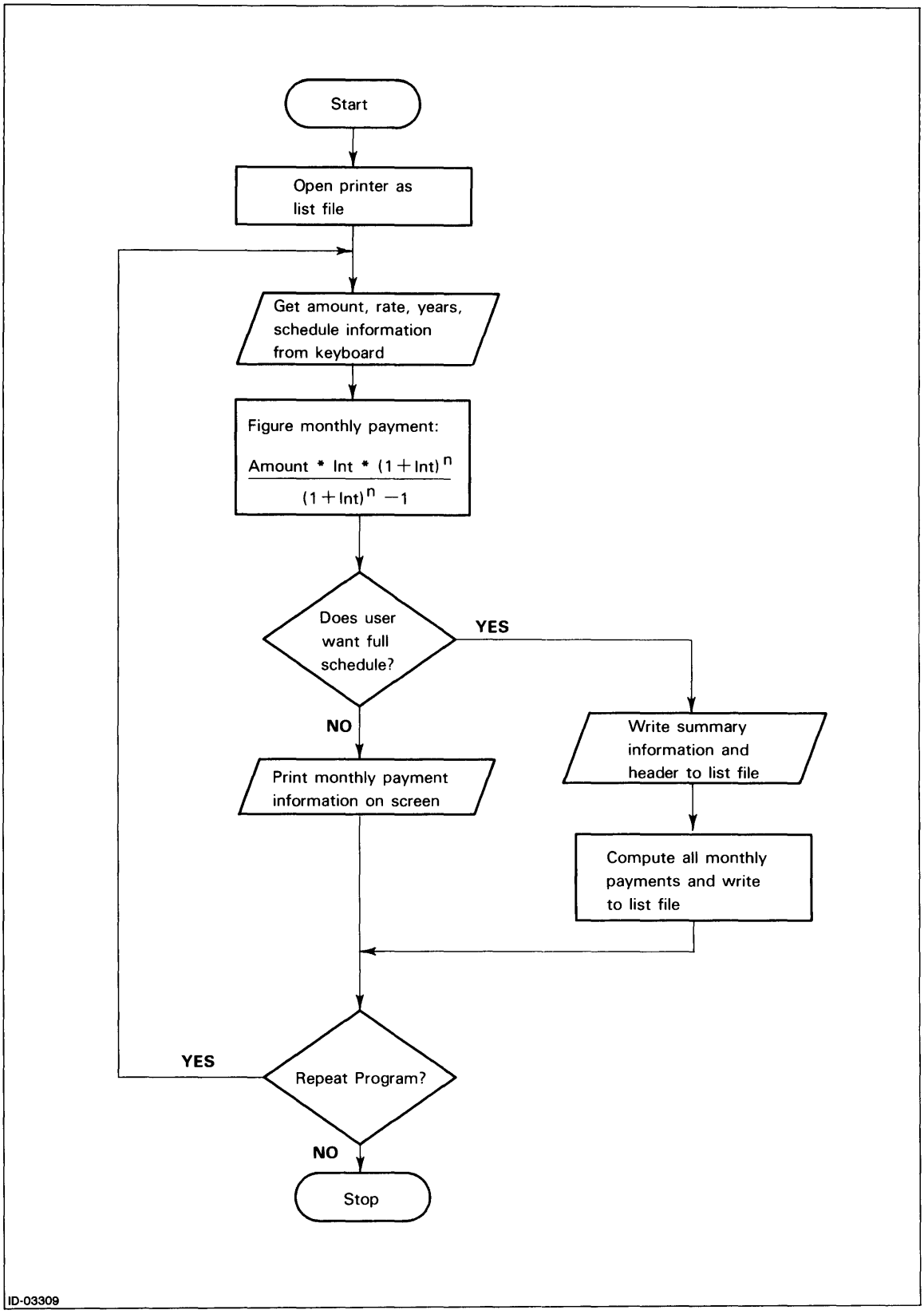


Figure 9-1. MORTGAGE Program Flow Chart (COBOL)

Writing the COBOL Source Program

If you want to create the MORTGAGE program, we suggest that you first make a directory for it. This will prevent conflicts with other programs that have the same name, and will encourage you to put all your COBOL programs in the same place. For example, type

```
) DIR/I ↓
) CRE/DIR COBOL_PROGRAMS ↓
) DIR COBOL_PROGRAMS ↓
```

Now, execute the SED or SPEED text editor:

```
) XEQ SED MORTGAGE.COB ↓
```

or

```
) XEQ SPEED/D MORTGAGE.COB ↓
```

Using the SED or SPEED text editor, type in the program as shown in Figure 9-2. Don't forget to insert a tab (TAB key or CTRL-I) as needed. (If you choose not to create a COBOL directory, use a filename other than MORTGAGE — for example, MORTGAGE.COBOL.)

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MORTGAGE.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT OUTFILE, ASSIGN TO PRINTER "@LIST".

DATA DIVISION.
FILE SECTION.
FD      OUTFILE, BLOCK CONTAINS 512 CHARACTERS.
01      OUTREC.
        02 OUR-PAYMT-NUM        PIC ZZZ9.
        02 FILLER               PIC X(6).
        02 OUT-MON-INT          PIC $(4)9.99.
        02 FILLER               PIC X(3).
        02 OUT-MON-PRIN         PIC $(6)9.99.
        02 FILLER               PIC X(3).
        02 OUT-BALANCE          PIC $(6)9.99.
        02 FILLER               PIC X(8).
        02 OUT-INT-TO-DATE      PIC $(6)9.99.
        02 FILLER               PIC X(10).

WORKING-STORAGE SECTION.
01      CRT-INPUTS.
        02 PRINCIPAL            PIC 9(6)V99.
        02 PERCENT              PIC 99V99.
        02 YEARS                PIC 99.
        02 FUNCTION             PIC 9.
        02 REPEAT-FLAG          PIC 9.

01      TEMPS.
        02 MONTHLY-INT-RATE     USAGE COMP-1.
        02 MONTHS               PIC 9(4).
        02 MONTHS-LEFT          PIC 9(4).
        02 MONTHLY-PAYMT        PIC 9(4)V99.
        02 LOAN-BAL             PIC 9(6)V99.
        02 INT-TO-DATE          PIC 9(6)V99.
        02 PAYMT-NUM            PIC 9(4), USAGE COMP.
        02 INT-PAYMT            PIC 9(4)V99.
        02 PRIN-PAYMT           PIC 9(4)V99.

01      SUMMARY-LINE 1.
        02 FILLER       PIC X(16), VALUE "Principal =     ".
        02 SUMMARY-PRIN, PIC $(6)9.99.
01      SUMMARY-LINE2.
        02 FILLER       PIC X(20), VALUE "Interest rate =     ".
        02 SUMMARY-RATE, PIC 9.9(4).
```

Figure 9-2. COBOL MORTGAGE Program with Errors (continues)

```
01      SUMMARY-LINE3.
        02 FILLER       PIC X(18), VALUE "Loan life =       ".
        02 SUMMARY-YEARS, PIC Z9.
        02 FILLER       PIC X(6), VALUE " years".
01      SUMMARY-LINE4.
        02 FILLER       PIC X(18), VALUE "Monthly payment = ".
        02 SUMMARY-PAYMT, PIC $(4)9.99.

01      HEADLINE        PIC X(80),
            VALUE " Num       Interest     Prin. Pay  Prin.                "
-                 " Bal.     Interest Paid to Date".

PROCEDURE DIVISION.
INIT.   OPEN OUTPUT OUTFILE.
OPERATOR.
        DISPLAY "Enter principal:   $" WITH NO ADVANCING.
        ACCEPT PRINCIPAL.
        DISPLAY "Interest rate (%): " WITH NO ADVANCING.
        ACCEPT PERCENT.
        COMPUTE MONTHLY-INT-RATE = PERCENT / 100 / 12.
        DISPLAY "Years to pay:      " WITH NO ADVANCING.
        ACCEPT YEARS.
        COMPUTE MONTHS = YEARS * 12.
        DISPLAY "Type 0 for Summary or 1 for Full Schedule:  "
                                WITH NO ADVANCING.
        ACCEPT FUNCTION.
        COMPUTE MONTHLY-PAYMT ROUNDED =
                PRINCIPAL * MONTHLY-INT-RATE *
                        (1 + MONTHLY-INT-RATE) ** MONTHS /
*               ---------------------------------------
                ((1 + MONTHLY-INT-RATE) ** MONTHS - 1).
        PERFORM SUMMARY-OUTPUT.
        IF FUNCTION NOT = 0,
                PERFORM DETAIL-OUTPUT.
        DISPLAY "Type 1 to repeat, 0 to stop: " WITH NO ADVANCING.
        ACCEPT REPEAT-FLAG.
        IF REPEAT-FLAG NOT = 0, GO TO OPERATOR.
        CLOSE OUTFILE.
        STOP RUN.

SUMMARY-OUTPUT.
        MOVE PRINCIPAL TO SUMMARY-PRIN.
        WRITE OUTREC FROM SUMMARY-LINE1 BEFORE ADVANCING 1.
        COMPUTE SUMMARY-RATE = PERCENT / 100.
        WRITE OUTREC FROM SUMMARY-LINE2 BEFORE ADVANCING 1.
        MOVE YEARS TO SUMMARY-YEARS.
        WRITE OUTREC FROM SUMMARY-LINE3 BEFORE ADVANCING 2.
        MOVE MONTHLY-PAYMT TO SUMMARY-PAYMT.
        WRITE OUTREC FROM SUMMARY-LINE4 BEFORE ADVANCING 2.
```

Figure 9-2. COBOL MORTGAGE Program with Errors (continued)

```
DETAIL-OUTPUT.
        MOVE PRINCIPAL TO LOAN-BAL.
        MOVE MONTHS TO MONTHS-LEFT.
        MOVE 0 TO INT-TO-DATE.
        WRITE OUTREC FROM HEADLINE BEFORE ADVANCING 2.
        MOVE SPACES TO OUTREC.
        PERFORM DO-DETAIL-LINE
                VARYING PAYMT-NUM FROM 1 BY 1
                        UNTIL PAYMT-NUM > MONTHS.
DO-DETAIL-LINE.
        COMPUTE PRIN-PAYMT ROUNDED =
                LOAN-BAL * MONTHLY-INT-RATE /
*               ----------------------------------------
                ((1 + MONTHLY-INT-RATE) ** MONTHS-LEFT - 1).
        SUBTRACT 1 FROM MONTHS-LEFT.
        COMPUTE INT-PAYMT = MONTHLY-PAYMT - PRIN-PAYMT.
        SUBTRACT PRIN-PAYMT FROM LOAN-BAL.
        ADD INT-PAYMT TO INT-TO-DATE.
        MOVE PAYMT-NUM TO OUT-PAYMT-NUM.
        MOVE INT-PAYMT TO OUT-MON-INT.
        MOVE PRIN-PAYMT TO OUT-MON-PRIN.
        MOVE LOAN-BAL TO OUT-BALANCE.
        MOVE INT-TO-DATE TO OUT-INT-TO-DATE.
        WRITE OUTREC BEFORE ADVANCING 2.
LAST-LINE.
```

Figure 9-2. COBOL MORTGAGE Program with Errors (concluded)

Compiling the COBOL Program

Having written the mortgage program, your next task is to compile it. Since the initial version undoubtedly has some errors, ask for a program listing. This listing will help identify the errors.

The command to compile your COBOL program is

```
) COBOL/L=@LPT MORTGAGE ↵
```

The /L=@LPT switch, which is optional, directs the compiler to provide a program listing, complete with errors, and send it to the line printer.

The compiler displays its banner, with the date, time, revision number and the specified options, and then it displays several error messages:

```
Error 475 severity 3 beginning on line 42 near 1
The parser has found an illegal construct in this data description
entry.  This may be due to a missing reserve word, a missing period, or a
misspelled word.

Error 335 severity 3 beginning on line 90 near SUMMARY-LINE1
The identifier 'SUMMARY-LINE1' is undefined.  Correct this error and recompile
this program.

Error 477 severity 3 beginning on line 90 near SUMMARY-LINE1
Missing identifier following FROM.

Warning 174 severity 1 beginning on line 90 near 1
Paragraph and Section names must start in margin A.

Error 335 severity 3 beginning on line 116 near OUT-PAYMT-NUM
The identifier 'OUT-PAYMT-NUM' is undefined.  Correct this error and recompile
this program.

Error 335 severity 3 beginning on line 117.
The identifier 'OUT-PAYMT-NUM' is undefined.  Correct this error and recompile
this program.
```

The error messages seem to imply a lot of errors. However, all the errors were caused by only two mistakes.

The first mistake is easy to find. On line 43, is *SUMMARY-LINE 1.* instead of *SUMMARY-LINE1.* The space informed the compiler that there were two data description entries.

The second error — flagged on line 117 — is less obvious. The error originated in the data division's file section, in line 13, where incorrect *OUR-PAYMT-NUM* appears instead of the correct *OUT-PAYMENT-NUM.* The alphabetical identifier listing, produced whenever you specify a listing file, can help identify this kind of error.

Knowing where the errors are, use a text editor to correct the MORTGAGE program. Correct lines 13 and 43, producing the following changed lines:

```
        02  OUT-PAYMT-NUM       PIC ZZZ9.

01      SUMMARY-LINE1.
```

Leave the text editor and enter the command to compile the program:

```
) COBOL MORTGAGE )
```

This time, there are no errors, and the CLI prompt appears when compilation is complete. Now, build the object file into an executable program file.

Creating the Program File

The command that produces an executable program file from your compiled program is

```
) CLINK MORTGAGE ↵
```

Your system may prompt you to include the notation ICALL32 on the Link command line (CLINK MORTGAGE ICALL32). ICALL32 is the interface to AOS/VS INFOS II.

The Link program executes and displays the following banner and messages on your screen:

```
*** AOS/VS COBOL LINK of MORTGAGE ***
LINK REVISION nnnnn ON  date  AT  time
OPTIONS ...
MORTGAGE.PR CREATED
```

Executing the COBOL Program

To execute your COBOL program, or any other one, simply type the XEQ command followed by the program name and press NEW LINE:

```
) XEQ MORTGAGE ↵
```

However, this time, instead of running, the MORTGAGE program displays an error message:

```
Error encountered during processing of file: @LIST

ERROR          21.

FILE DOES NOT EXIST.

ERROR          21.
```

A fatal runtime error! And we find the CLI running on the terminal. The COBOL message *FILE DOES NOT EXIST* indicates that the file @LIST is missing.

The source of the problem is the list file. The program tried to open the file PRINTER @LIST, an instruction specifying a list file.

However, @LIST is a generic file; in itself it does nothing but allow you to write code independent of a device. @LIST is a pointer to a file that you establish at runtime with the LISTFILE command. You can set it to any legal filename you want, but you must set it before a program can access it. Thus, when the program tried to open @LIST, the system couldn't find the file, so it stopped the program with a runtime error message.

To correct this, simply set the @LIST to the terminal, whose filename is @CONSOLE:

```
) LISTFILE @CONSOLE ↵
```

Having set @LIST to the terminal, you can successfully execute the program:

```
) XEQ MORTGAGE ⤸
```

This time, the MORTGAGE program prompts for statistics on the loan:

```
Enter principal:
```

Now enter plausible figures and engage in the following dialog with the MORTGAGE program:

```
Enter principal:  $   80000 ⤸
Interest rate (%):    12.5 ⤸
Years to pay:         30 ⤸

Type 0 for Summary or 1 for Full Schedule:     0 ⤸

Principal =             $80000.00
Interest rate =            0.1250
Loan Life =              30 years

Monthly payment=         $853.81

Type 1 to repeat, 0 to stop.      1 ⤸
```

You typed 1 to run the MORTGAGE program again. This time you'll give the same figures, but request a full schedule:

```
Enter principal:  $   80000 ⤸
Interest rate (%):    12.5 ⤸
Years to pay:         30 ⤸

Type 0 for Summary or 1 for Full Schedule:      1 ⤸

Principal =             $80000.00
Interest rate =            0.1250
Loan Life =              30 years

Monthly payment=         $853.81

 Num       Interest     Prin. Pay  Prin. Bal.     Interest Paid to Date

   1       $833.34       $20.47    $79979.53           $833.34

   2       $833.12       $20.69    $79958.84          $1666.46
.
.
.
```

It works! Stop it with CTRL-C, CTRL-B:

```
CTRL-C   CTRL-B

*ABORT*
ERROR: CONSOLE INTERRUPT
ERROR: FROM PROGRAM
xeq mortgage
```

Now that we know it works, let's set the list file to a disk file instead of the terminal, and execute the program again, supplying the same figures:

```
) LISTFILE MORTGAGE.OUTPUT )
) XEQ MORTGAGE )

Enter principal: $      80000 )
Interest rate (%):      12.5 )
Years to pay:           30 )

Type 0 for Summary or 1 for Full Schedule:      1 )
```

This time, you see nothing because the MORTGAGE program is writing everything to the file MORTGAGE.OUTPUT, not to your screen. After a minute or so, the MORTGAGE program prompts for directions, and we type 0 to stop the program:

```
Type 1 to repeat, 0 to stop.        0 )
```

The MORTGAGE program terminates, closing the list file MORTGAGE.OUTPUT. MORTGAGE.OUTPUT will remain in the working directory. While it remains the list file, each full schedule we ask for when we run MORTGAGE will be appended to it. To print the list file, type

```
) QPR MORTGAGE.OUTPUT )
QUEUED,   SEQ=n
```

Checking the printer, we find the original summary, and then the full schedule, which is about eight pages long. Figure 9-3 shows the beginning of the full schedule. The interest total at the end (which the Figure 9-3 doesn't show but your listing will) is pretty appalling.

Preceding the summary and the full schedule is a printed header that describes our username, output file pathname, and date.

```
Principal =      $80000.00

Interest rate =     0.1250

Loan life =       30 years


Monthly payment =  $853.81

 Num       Interest     Prin. Pay  Prin. Bal.     Interest Paid to Date

   1       $833.34       $20.47    $79979.53          $833.34

   2       $833.12       $20.69    $79958.84         $1666.46

   3       $832.91       $20.90    $79937.94         $2499.37

   4       $832.69       $21.12    $79916.82         $3332.06
   .          .            .          .                .
   .          .            .          .                .
   .          .            .          .                .
  10       $831.34       $22.47    $79785.41         $8323.51

  11       $831.10       $22.71    $79762.70         $9154.61

  12       $830.87       $22.94    $79739.76         $9985.48
```

Figure 9-3. Beginning of the Full Payment Schedule from the MORTGAGE Program (COBOL)

The MORTGAGE program runs successfully, and we don't need to do any more work on it.

Remember the List File

The list file is still set to MORTGAGE.OUTPUT. But the next time you log on, it will be set to the dummy filename @LIST, as when we tried to compile the MORTGAGE program. So, the next time you log on, if you run MORTGAGE and receive a ***FILE DOES NOT EXIST*** error message, simply set @LIST to the device or disk file you want. For example, type

```
) LISTFILE @CONSOLE ↵
```

or

```
) LISTFILE MORTGAGE.OUT ↵
```

You can see the flexibility of the list file, specified in the program on line 7, as ...PRINTER "@LIST". If you hadn't added the "@LIST", the program output would have been restricted to the line printer. Having stipulated @LIST, you can select your output file at will from the CLI.

Using the SWAT® Debugger

Data General's SWAT® debugger, if your system has it, can really ease the debugging phase of program development. To use SWAT, compile your program with the /DEBUG switch:

```
) COBOL/DEBUG MORTGAGE ↵
```

Then link the object modules using the /DEBUG switch:

```
) CLINK/DEBUG MORTGAGE ↵
```

And start up the program in the debugger:

```
) SWAT MORTGAGE ↵

AOS/VS SWAT Revision...
```

In SWAT, you can set breakpoints by listing a line number with the BREAKPOINT command, display source lines to check logic with the LIST command, start or run the program with the CONTINUE command, examine variables at breakpoints with the TYPE command, receive assistance at any point with the HELP command, and leave SWAT with the BYE command.

A SWAT session with the MORTGAGE program might involve the following dialog:

```
> BREAK 80 ↵
Set at :MORTGAGE: 80

> CON ↵
Enter principal: $      80000 ↵
Interest rate (%):      12.5 ↵
Years to pay:           30 ↵

Type 0 for Summary or 1 for Full Schedule:      0 ↵
```

The program stops at line 80 where you set the breakpoint. Now you can check program values.

```
Breakpoint trap at :MORTGAGE:80

> TYPE MONTHLY-PAYMT ⤸
853.81
> TYPE PRINCIPAL ⤸
80000.
> BYE ⤸
This Swat session is terminating.
```

In the SWAT session the program stops at line 80, where you set a breakpoint. There you checked the program values of two variables: MONTHLY-PAYMT and PRINCIPAL. Since they're both reasonable, you close the debugging session and exit with the BYE command.

What Next?

If you want to practice with another language, or review earlier material, continue to the appropriate chapter. Or, you can start writing your own COBOL programs, using the *COBOL Reference Manual (AOS/VS).*

For details on COBOL and SWAT manuals, and user documentation in general, refer to Chapter 16.

End of Chapter

Chapter 10
Interactive COBOL Programming

This chapter describes how to develop, run, and debug an Interactive COBOL program under AOS/VS. If you're totally unfamiliar with Interactive COBOL, we recommend that you turn to a language reference, listed in Chapter 16. Once you have a sense of the language, return to this chapter for an example of AOS/VS program development.

Program Development Sequence

These are the steps you follow to write an Interactive COBOL program:

1. Write and, if necessary, edit the Interactive COBOL program source file with one of the text editors, SED or SPEED:

   ```
   ) xeq sed filename ↓
   ```

   or

   ```
   ) xeq speed/d filename ↓
   ```

2. Compile the source file with the compile macro supplied by Data General:

   ```
   ) icobol/l=@lpt filename ↓
   ```

   Add a /D switch if you plan to debug the program.

3. If there are any errors from the compiler, go to step 1 and fix them with your chosen editor.
4. Execute the program from the CLI with the command line,

   ```
   ) icx filename ↓
   ```

5. If the program runs the way you want it to, go to step 8.
6. Identify logic errors via runtime error messages or erroneous output. If your system has the Interactive COBOL debugger, debug the program with the command,

   ```
   ) icdebug filename ↓
   ```

7. Go to step 1 and fix the incorrect code with your chosen text editor.
8. You're done! You can go on to write your own programs under AOS/VS.

The Interactive COBOL Example Program

The program in this chapter calculates home mortgage payments. It can also produce a month-by-month schedule of principal and interest. Figure 10-1 is the program flow chart, and Figure 10-2 is the Interactive COBOL program itself.

The mortgage formulas are those used by U.S. banks. Different formulas are used outside the United States. If you live outside the United States, treat the formula parts of the program as examples only. (Later, you might want to replace the existing formulas with one appropriate to your nation.)

If you try the FORTRAN program in Chapter 11, which is also a mortgage program, you'll see startling differences between it and the Interactive COBOL program. The Interactive COBOL program defines all variables ("data names") and record structures at the beginning; and it also has divisions, sections, sentences, and clauses. It is much more structured and much longer than its FORTRAN counterpart.

To compile and run Interactive COBOL programs, your system must have the appropriate compiler files and libraries. The directory that holds these, often directory :UTIL:ICOBOL, must be in your search list. Set your search list by typing

```
) sea :util:ICOBOL [!sea] ↵
```

Directory :UTIL:ICOBOL will remain on your search list until you reset it or log off.

Even if you decide not to copy the Interactive COBOL program, you should, in addition to reading the text, examine Figures 10-1 and 10-2 before writing original programs.

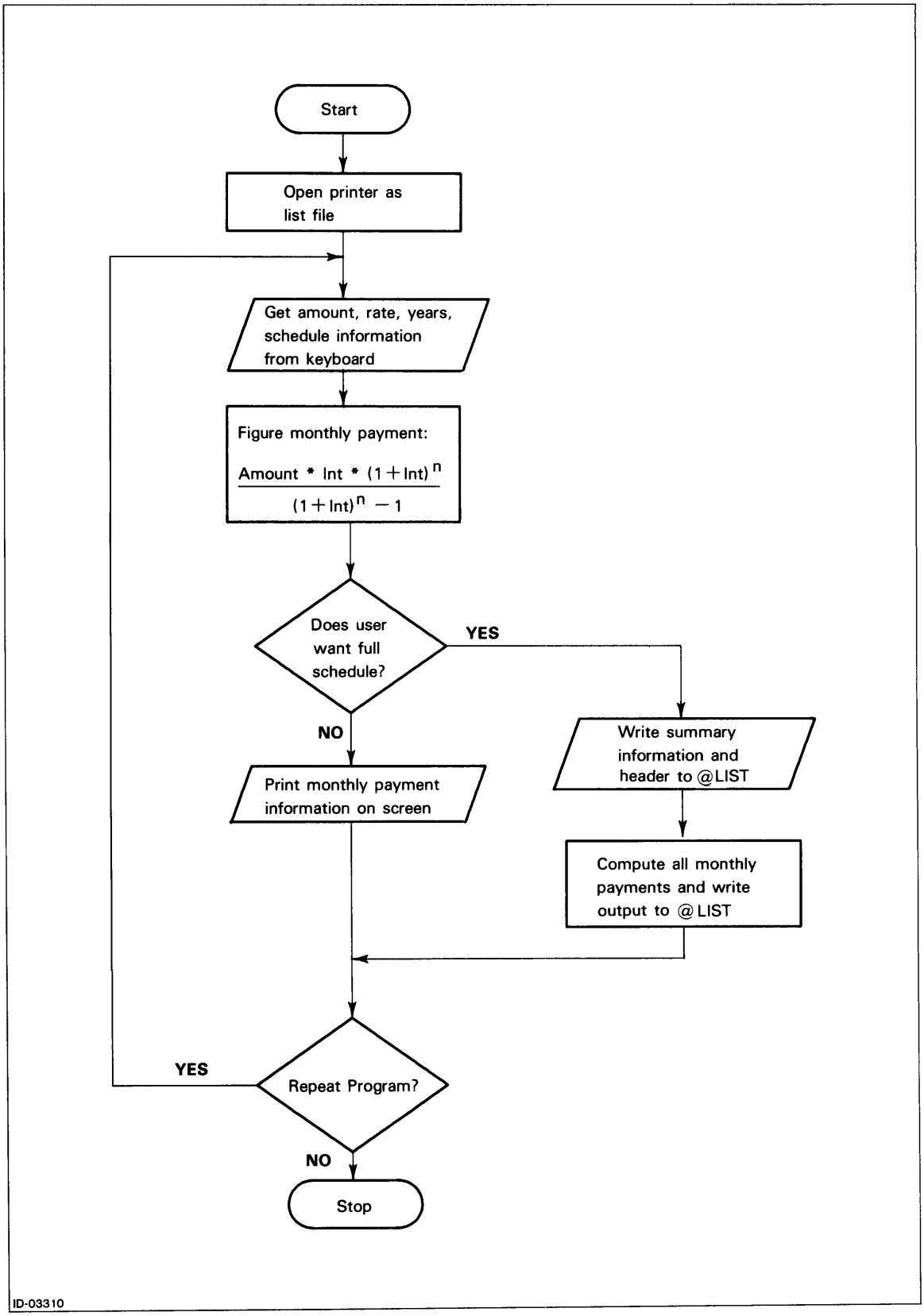


Figure 10-1. MORTGAGE Program Flow Chart (Interactive COBOL)

Writing the Interactive COBOL Source Program

If you want to copy the MORTGAGE program, we suggest that you first create a directory for it. This will prevent conflicts with other programs that have the same name and will encourage you to put all your Interactive COBOL programs in the same directory. For example, type

```
) dir/i ⤸
) cre/dir icobol ⤸
) dir icobol ⤸
```

Now, from the working directory ICOBOL, run the SED or SPEED text editor:

```
) xeq sed mortgage ⤸
```

or

```
) xeq speed/d mortgage ⤸
```

Using the SED or SPEED text editor, type in the program, complete with errors, according to Figure 10-2. Don't forget to insert tabs (TAB key or CTRL-I) as needed. (If you chose not to create an ICOBOL directory, use a more specific filename than MORTGAGE — for example MORTGAGE.COBOL.)

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    MORTGAGE.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT OUTFILE ASSIGN TO PRINTER, "@LIST".

DATA DIVISION.
FILE SECTION.
FD      OUTFILE
        LABEL RECORDS ARE OMITTED.
01      OUTREC.
        02 OUR-PAYMT-NUM        PIC ZZZ9.
        02 FILLER               PIC X(6).
        02 OUT-MON-INT          PIC $(4)9.99.
        02 FILLER               PIC X(3).
        02 OUT-MON-PRIN         PIC $(6)9.99.
        02 FILLER               PIC X(3).
        02 OUT-BALANCE          PIC $(6)9.99.
        02 FILLER               PIC X(8).
        02 OUT-INT-TO-DATE      PIC $(6)9.99.
        02 FILLER               PIC X(10).

WORKING-STORAGE SECTION.
01      CRT-INPUTS.
        02 PRINCIPAL            PIC 9(6)V99.
        02 PERCENT              PIC 99V99.
        02 YEARS                PIC 99.
        02 FUNCTION             PIC 9.
        02 REPEAT-FLAG          PIC 9.

01      TEMPS.
        02 MONTHLY-INT-RATE     PIC 99V9999.
        02 MONTHS               PIC 9(4).
        02 MONTHS-LEFT          PIC 9(4).
        02 MONTHLY-PAYMT        PIC 9(4)V99.
        02 LOAN-BAL             PIC 9(6)V99.
        02 INT-TO-DATE          PIC 9(6)V99.
        02 PAYMT-NUM            PIC 9(4), USAGE COMP.
        02 INT-PAYMT            PIC 9(4)V99.
        02 PRIN-PAYMT           PIC 9(4)V99.

01      SUMMARY-LINE 1.
        02 FILLER       PIC X(16), VALUE "PRINCIPAL =     ".
        02 SUMMARY-PRIN, PIC $(6)9.99.
        02 FILLER       PIC X(50), VALUE SPACES.
01      SUMMARY-LINE2.
        02 FILLER       PIC X(20), VALUE "INTEREST RATE =     ".
        02 SUMMARY-RATE, PIC 9.9(4).
        02 FILLER       PIC X(50), VALUE SPACES.
```

Figure 10-2. Interactive COBOL MORTGAGE Program with Errors (continues)

```
01      SUMMARY-LINE3.
        02 FILLER       PIC X(18), VALUE "LOAN LIFE =       ".
        02 SUMMARY-YEARS, PIC Z9.
        02 FILLER       PIC X(6), VALUE " YEARS".
        02 FILLER       PIC X(50), VALUE SPACES.
01      SUMMARY-LINE4.
        02 FILLER       PIC X(18), VALUE "MONTHLY PAYMENT = ".
        02 SUMMARY-PAYMT, PIC $(4)9.99.
01      HEADLINE        PIC X(80),
            VALUE " NUM       INTEREST     PRIN. PAY  PRIN.        "
-                " BAL      INTEREST PAID TO DATE".

PROCEDURE DIVISION.
INIT.   OPEN OUTPUT OUTFILE.
OPERATOR.
        DISPLAY "ENTER PRINCIPAL:   $" WITH NO ADVANCING.
        ACCEPT PRINCIPAL.
        DISPLAY "INTEREST RATE (%): " WITH NO ADVANCING.
        ACCEPT PERCENT.
        COMPUTE MONTHLY-INT-RATE ROUNDED = PERCENT / 100 / 12.
        DISPLAY "YEARS TO PAY:      " WITH NO ADVANCING.
        ACCEPT YEARS.
        COMPUTE MONTHS = YEARS * 12.
        DISPLAY "FUNCTION (0=SUMMARY, 1=FULL SCHEDULE): "
                                WITH NO ADVANCING.
        ACCEPT FUNCTION.
        COMPUTE MONTHLY-PAYMT ROUNDED =
                PRINCIPAL * MONTHLY-INT-RATE *
                        (1 + MONTHLY-INT-RATE) ** MONTHS /
*               ---------------------------------------
                ((1 + MONTHLY-INT-RATE) ** MONTHS - 1).
        PERFORM SUMMARY-OUTPUT.
        IF FUNCTION NOT = 0,
                PERFORM DETAIL-OUTPUT.
        DISPLAY "TYPE 1 TO REPEAT, 0 TO STOP: " WITH NO ADVANCING.
        ACCEPT REPEAT-FLAG.
        IF REPEAT-FLAG NOT = 0, GO TO OPERATOR.
        CLOSE OUTFILE.
        STOP RUN.

SUMMARY-OUTPUT.
        MOVE PRINCIPAL TO SUMMARY-PRIN.
        WRITE OUTREC FROM SUMMARY-LINE1 BEFORE ADVANCING 1.
        COMPUTE SUMMARY-RATE = PERCENT / 100.
        WRITE OUTREC FROM SUMMARY-LINE2 BEFORE ADVANCING 1.
        MOVE YEARS TO SUMMARY-YEARS.
        WRITE OUTREC FROM SUMMARY-LINE3 BEFORE ADVANCING 2.
        MOVE MONTHLY-PAYMT TO SUMMARY-PAYMT.
        WRITE OUTREC FROM SUMMARY-LINE4 BEFORE ADVANCING 2.
```

Figure 10-2. Interactive COBOL MORTGAGE Program with Errors (continued)

```
DETAIL-OUTPUT.
        MOVE PRINCIPAL TO LOAN-BAL.
        MOVE MONTHS TO MONTHS-LEFT.
        MOVE 0 TO INT-TO-DATE.
        WRITE OUTREC FROM HEADLINE BEFORE ADVANCING 2.
        MOVE SPACES TO OUTREC.
        PERFORM DO-DETAIL-LINE
                VARYING PAYMT-NUM FROM 1 BY 1
                        UNTIL PAYMT-NUM > MONTHS.

DO-DETAIL-LINE.
        COMPUTE PRIN-PAYMT ROUNDED =
                LOAN-BAL * MONTHLY-INT-RATE /
*               ------------------------------------------
                ((1 + MONTHLY-INT-RATE) ** MONTHS-LEFT - 1).
        SUBTRACT 1 FROM MONTHS-LEFT.
        COMPUTE INT-PAYMT = MONTHLY-PAYMT - PRIN-PAYMT.
        SUBTRACT PRIN-PAYMT FROM LOAN-BAL.
        ADD INT-PAYMT TO INT-TO-DATE.
        MOVE PAYMT-NUM TO OUT-PAYMT-NUM.
        MOVE INT-PAYMT TO OUT-MON-INT.
        MOVE PRIN-PAYMT TO OUT-MON-PRIN.
        MOVE LOAN-BAL TO OUT-BALANCE.
        MOVE INT-TO-DATE TO OUT-INT-TO-DATE.
        WRITE OUTREC BEFORE ADVANCING 2.
LAST-LINE.
```

Figure 10-2. Interactive COBOL MORTGAGE Program with Errors (concluded)

Compiling the Interactive COBOL Program

Having written the mortgage program, you now compile it. Since the initial version undoubtedly has some errors, ask for a program listing. This listing will help identify the errors.

The command to compile your Interactive COBOL program is

```
) icobol/l=@lpt mortgage )
```

The /L=@LPT switch directs the compiler to provide a program listing, with errors, and send it to the line printer for a hardcopy.

The compiler displays its banner, noting the date, time, revision number, and program name. Then it displays the following error messages:

```
         COMPILED  date  AT time  AOS/VS INTERACTIVE COBOL REV. n
   2     PROGRAM-ID.    MORTGAGE.
  45             02 FILLER       PIC X(16), VALUE "PRINCIPAL =     ".
 ERROR:  UNRECOGNIZABLE WORD
  35             WRITE OUTREC FROM SUMMARY-LINE1 BEFORE ADVANCING 1.
 ERROR:  DATA NAME UNDEFINED SUMMARY-LINE1
   2             MOVE PAYMT-NUM TO OUT-PAYMT-NUM.

 ERROR:  DATA NAME UNDEFINED OUT-PAYMT-NUM
.
.
.
COMPILED WITH ERRORS
```

The messages seem to suggest a lot of errors. But all errors were caused by only two mistakes. The first mistake is easy to find.

On line 44, find the incorrect SUMMARY-LINE 1. that should have been SUMMARY-LINE1. The space informed the compiler that there were two data description entries. This error also affected the definition of SUMMARY-PRIN on line 46, causing the the compiler to flag SUMMARY-PRIN later in the program.

The second error — flagged on line 117 — is less obvious. The error originated in the data division's file section, in line 14, where the wording OUR-PAYMT-NUM is incorrect; it should be OUT-PAYMENT-NUM. The alphabetical identifier listing, produced whenever you specify a listing file, can help identify this kind of error.

Next, using the SED or SPEED text editor, edit the MORTGAGE program. Correct lines 14 and 44, producing — respectively — the following changed lines:

```
        02 OUT-PAYMT-NUM        PIC ZZZ9.

01      SUMMARY-LINE1.
```

Leave the editor and type the compiler command line again, this time without the /L=@LPT switch:

```
) icobol mortgage ⤸
```

Now there are no errors, and the CLI prompt appears when the compilation is complete. You are ready to execute the program.

Executing the Interactive COBOL Program

To execute our Interactive COBOL program, enter the runtime command ICX and the program it should run:

```
) icx mortgage ⤸
```

But instead of running the MORTGAGE program, the system displays the following message:

```
*ERROR: Fatal COBOL Program I/O Error.          COBOL PC=n
ERROR: FROM PROGRAM
X/S, ICX, mortgage
```

A runtime error occurred, causing the program to stop. And we find the CLI running on the console. The message indicates a missing file.

The source of the problem is the list file. On line 7 of the MORTGAGE program, the program opened an output file, PRINTER, and assigned it to the generic file @LIST. A generic file is a CLI file that allows you to write programs that are independent of devices. @LIST acts as a pointer to a file that you must set before a program can access it. At runtime you can set @LIST to any legal filename you want. Thus, when the program tried to open @LIST, the system couldn't find the file and aborted the program with a runtime error message.

To correct this, simply set the list file to the terminal, whose name is @CONSOLE:

```
) listfile @console ⤸
```

Having set @LIST to the terminal, re-execute the program:

```
) icx mortgage ⤸
```

This time the MORTGAGE program prompts for the principal:

```
ENTER PRINCIPAL:
```

You now enter a plausible figure, such as $50,000, and engage in the following dialog with the MORTGAGE program:

```
ENTER PRINCIPAL:    $  50000 ⤸
INTEREST RATE (%):      13.5 ⤸
YEARS TO PAY:           25 ⤸
```

The program presents you with a choice of displaying a full schedule of monthly payments or a summary of the payments:

```
FUNCTION (0=SUMMARY, 1=FULL SCHEDULE):   0 ⤸
```

Choose a summary the first time. The program prints the following:

```
PRINCIPAL =          $50000.00
INTEREST RATE =      0.1350
LOAN LIFE =          25 YEARS

MONTHLY PAYMENT=     $585.11
```

And the program gives you the option to repeat or stop:

```
TYPE 1 TO REPEAT, 0 TO STOP.    1 ⤸
```

Type 1 to run it again, and enter the same figures. But this time, when prompted, request a full schedule:

```
ENTER PRINCIPAL:     $  50000 ↵
INTEREST RATE (%):       13.5 ↵
YEARS TO PAY:            25 ↵

FUNCTION (0=SUMMARY, 1=FULL SCHEDULE:     1 ↵
PRINCIPAL =            $50000.00
INTEREST RATE =        0.1350
LOAN LIFE =            25 YEARS

MONTHLY PAYMENT=       $585.11

 NUM       INTEREST     PRIN. PAY  PRIN. BAL      INTEREST PAID TO DATE

   1       $565.00        $20.11   $49979.89           $565.00

   2       $564.78        $20.33   $49959.56          $1129.78

   3       $564.55        $20.56   $49939.00          $1694.33

   .
   .
   .
```

The program works, so stop it with CTRL-C, CTRL-B:

```
CTRL-C   CTRL-B


***Program*Stopped*By*Console*Interrupt.***  COBOL PC=n
```

And set the list file to a disk file, MORTGAGE.OUTPUT, execute the program again, entering the same figures:

```
) listfile mortgage.output ↵
) icx mortgage ↵


ENTER PRINCIPAL:     $  50000 ↵
INTEREST RATE (%):       13.5 ↵
YEARS TO PAY:            25 ↵
```

Again the program presents you with a choice of a full schedule of monthly payments or a summary:

```
FUNCTION (0=SUMMARY, 1=FULL SCHEDULE):    1 ↵
```

This time, you see nothing on your screen because the MORTGAGE program is writing everything to the disk file MORTGAGE.OUTPUT. After a minute or so, the program prompts for directions:

```
TYPE 1 to REPEAT, 0 to STOP.    0 )
)
```

The MORTGAGE program terminates, closing the list file MORTGAGE.OUTPUT. MORTGAGE.OUTPUT will remain in the working directory. While it remains the list file, each full schedule you ask for when you run MORTGAGE will be appended to it. To print MORTGAGE.OUTPUT, type

```
) qpr mortgage.output )
QUEUED, \ SEQ=n
```

Checking the printer, you'll find the original summary and the full mortgage payment schedule, which is about eight pages long. Figure 10-3 shows the beginning of the full schedule. The interest total at the end (which the Figure 10-3 doesn't show, but your listing will) is pretty appalling.

Preceding the summary and the full schedule is a printed header that describes our username, output file pathname, and date.

The MORTGAGE program works well and there's no need to make further modifications to it.

```
PRINCIPAL =       $50000.00
INTEREST RATE =       0.1350
LOAN LIFE =         25 YEARS

MONTHLY PAYMENT =  $585.11

 NUM        INTEREST      PRIN. PAY   PRIN. BAL      INTEREST PAID TO DATE

   1        $565.00         $20.11    $49979.89            $565.00

   2        $564.78         $20.33    $49959.56           $1129.78

   3        $564.55         $20.56    $49937.00           $1694.33

   4        $564.32         $20.79    $49918.21           $2258.65
   .           .               .         .                    .
   .           .               .         .                    .
   .           .               .         .                    .
   9        $563.11         $22.00    $49810.65           $5076.64

  10        $562.61         $22.50    $49765.90           $5639.50
```

Figure 10-3. Beginning of the Full Payment Schedule from the MORTGAGE Program (Interactive COBOL)

Remember the List File

The list file is still set to MORTGAGE.OUTPUT. But the next time you log on, the list file will have the default setting of @LIST, as when you first tried to run the MORTGAGE program. The next time you log on, if you execute the MORTGAGE program and receive a *FILE DOES NOT EXIST* message, simply set @LIST to the device you want:

```
) listfile @console ↓
```

or

```
) listfile mortgage.out ↓
```

You can see the flexibility of the list file, specified in the program on line 7, as PRINTER "@LIST." If the program hadn't specified "@LIST", all output would have been restricted to the line printer. When you stipulate "@LIST", you can select your output file at will from the CLI.

On Using the Interactive COBOL Debugger

Data General's Interactive COBOL debugger can really ease the debugging phase of program development. The debugger requires that breakpoints be set at the beginning or the end of a paragraph. Therefore, to use the debugger, you must execute a text editor and create paragraph heads, which serve as breakpoints, at problematic points in the source file.

For example, let's insert an additional paragraph name in the PROCEDURE DIVISION of the MORTGAGE program to test for the value of MONTHLY-PAYMT.

Execute either SED or SPEED and locate the MONTHLY-PAYMT calculation in the PROCEDURE DIVISION. Insert the paragraph head: DEBUG-01. on the line after the computation, as shown below.

```
PROCEDURE DIVISION.
INIT.    OPEN OUTPUT OUTFILE.
OPERATOR.
        DISPLAY "ENTER PRINCIPAL:    $" WITH NO ADVANCING.
        .
        .
        .
        COMPUTE MONTHLY-PAYMT ROUNDED =
                PRINCIPAL * MONTHLY-INT-RATE *
                        (1 + MONTHLY-INT-RATE) ** MONTHS /
*               ------------------------------------
                ((1 + MONTHLY-INT-RATE) ** MONTHS - 1).

DEBUG-01.

        PERFORM SUMMARY-OUTPUT.
        .
        .
        .
```

Leave the editor, and now that the breakpoint is in the program, recompile with the /D (debug) switch:

```
) icobol/d mortgage ⤶
```

And start up the program in the debugger:

```
) icdebug mortgage ⤶

!
```

The debugger displays its ! prompt.

In the Interactive COBOL debugger, you can execute or continue the program with the RUN command, stop at breakpoints with the TRAP command, remove breakpoints with the CLEAR command, examine variables at breakpoints with the DISPLAY command, list breakpoints with the LIST command, and exit with the STOP command.

For example, an Interactive COBOL debugging session with MORTGAGE might involve the following dialog:

```
! trap debug-01 ⤶

*ERROR: Illegal command
```

The debugger requires uppercase, so the user re-enters the command:

```
! TRAP DEBUG-01 ⤶
! RUN ⤶

ENTER PRINCIPAL:     $  50000 ⤶
INTEREST RATE (%):       13.5 ⤶
YEARS TO PAY:            25 ⤶
FUNCTION (0 =SUMMARY, 1=FULL SCHEDULE):     0 ⤶

DEBUG-01   Trap Point At Start
! DISPLAY MONTHLY_PAYMT ⤶
585.11
! DISPLAY PRINCIPAL of CRT-INPUTS ⤶
50000.00
! RUN ⤶
TYPE 1 TO REPEAT, 0 TO STOP.     0 ⤶
```

Because you set a breakpoint set at DEBUG-01, the program runs to the calculation of monthly payment and stops. There you check the values in MONTHLY_PAYMT and PRINCIPAL. Both are reasonable, and the program runs to completion.

What Next?

If you want to learn about another language or review earlier material, continue to the appropriate chapter. You can also start writing your own Interactive COBOL programs, using the Interactive COBOL manuals listed in Chapter 16, Documentation Guide.

End of Chapter

Chapter 11
FORTRAN 77 Programming

This chapter describes how to develop and run a FORTRAN 77 program under AOS/VS. If you're totally unfamiliar with FORTRAN 77, we recommend that you turn to a language reference, listed in Chapter 16. Once you have a sense of the language, return to this chapter for an example of AOS/VS program development.

Program Development Sequence

These are the steps you follow to create a program in AOS/VS FORTRAN 77 (F77):

1. Create or edit an F77 source file with the SED or SPEED text editor:

   ```
   ) xeq sed filename.f77 ↵
   ```

   or

   ```
   ) xeq speed/d filename.f77 ↵
   ```

   With your chosen editor, type in the FORTRAN statements and comments that make up the program.

2. Compile the source file with the macro supplied by Data General:

   ```
   ) f77 filename ↵
   ```

   The compile command line can also include the /L=@LPT switch and the /DEBUG switch, described later.

3. If there are any errors from the compiler, return to step 1 and fix the incorrect statement(s). If there are no compilation errors, go to step 4.

4. Link the object file to produce an executable program:

   ```
   ) f77link filename ↵
   ```

   This command line can include the /DEBUG switch, described later.

5. Execute the program with the CLI command,

   ```
   ) xeq filename ↵
   ```

   If the program runs as you want it to, go to step 8.

6. Identify logic errors using runtime error messages or erroneous output. Or, if you included /DEBUG switches in steps 2 and 4 and your system has the SWAT interactive debugger, debug the program with the command,

   ```
   ) swat filename ↵
   ```

7. Return to step 1 if there are erroneous statement(s).

8. You're done! You can go on to write your own FORTRAN programs under AOS/VS.

This chapter guides you through all the steps you need to write and execute an F77 program.

The FORTRAN Program Example

The sample program calculates home mortgage payments; it also produces a schedule of monthly principal and interest. The program uses only ordinary arithmetic operations, calls no subroutines, and is less than a page long. The program uses two formulas. You need not understand how these formulas work to understand this example of writing, compiling, and executing an F77 program.

The mortgage formulas are those used by U.S. banks. Different formulas are used outside the United States. If you live outside the United States, treat the formula parts of the program as examples only. (Later, you might want to replace the existing formulas with ones appropriate to your nation.)

Figure 11-1 shows a flow chart for this program. Figure 11-2 shows the initial version of the program MORTGAGE.F77 itself, complete with errors.

Even if you decide not to copy this program, you should at least examine Figures 11-1 and 11-2 before reading the rest of the chapter.

To compile, link, and run FORTRAN 77 programs, your system must have the appropriate compiler files and libraries. The directory that holds these — often directory :UTIL:F77 or :F77 — must be in your search list. (The directory is named by the person at your site who installed F77.) To set your search list, assuming the FORTRAN directory is subordinate to :UTIL, type

```
) sea :util:F77  [!sea] ↓
```

The directory remains on your search list until you delete it or log off the system.

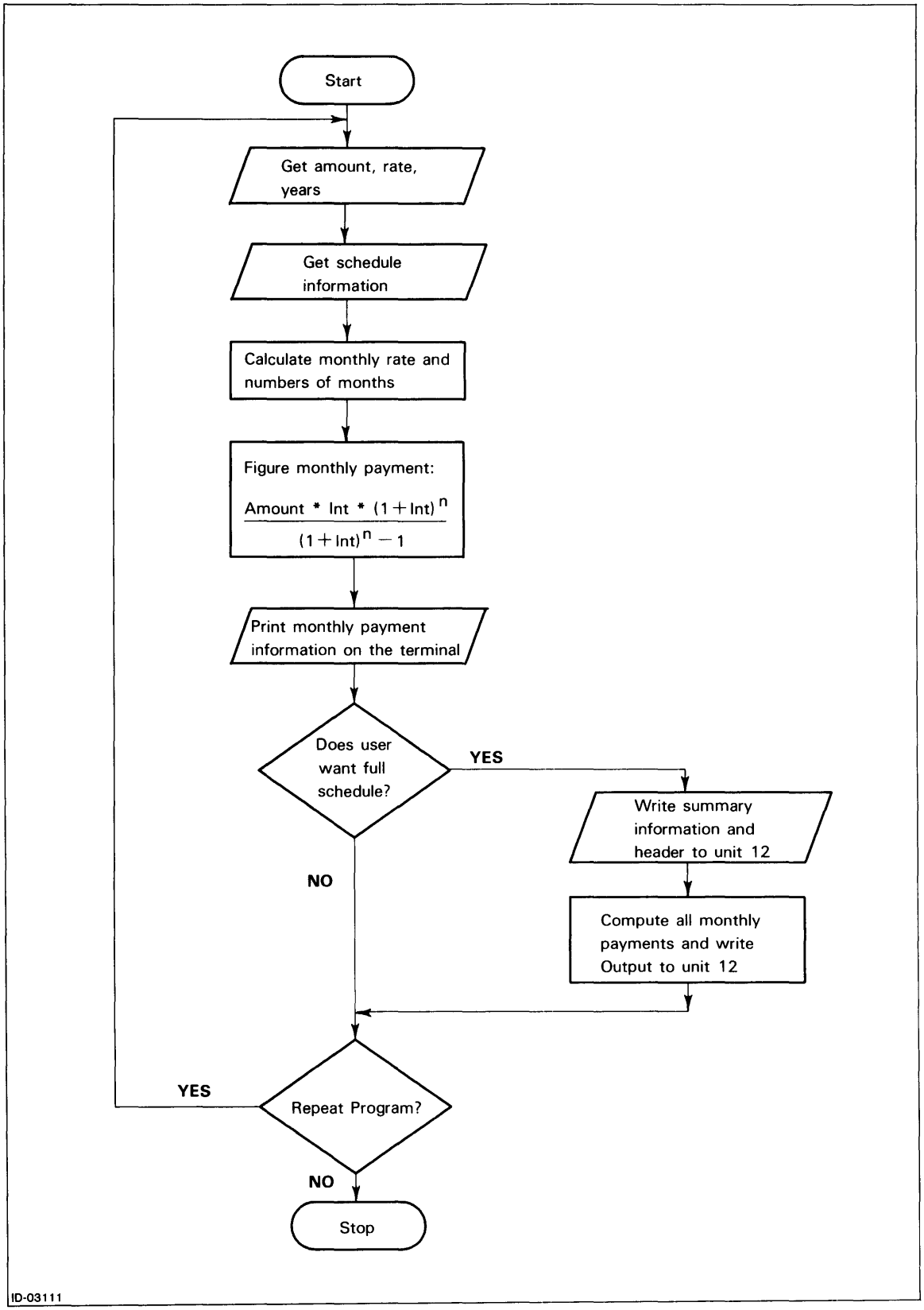


Figure 11-1. MORTGAGE Program Flow Chart (FORTRAN 77)

Writing the FORTRAN Program

If you want to try the MORTGAGE program, we suggest that you first create a directory for it. This will prevent conflicts with other programs that have the same name and will encourage you to place all your FORTRAN programs in the same place. For example, type

```
) dir/i ⤸
) cre/dir FORTRAN ⤸
) dir FORTRAN ⤸
```

Now, enter the SED or SPEED text editor, and create the source file. We suggest that the source filename end with the conventional FORTRAN 77 suffix, .F77.

```
) xeq sed mortgage.f77 ⤸
```

or

```
) xeq speed/d mortgage.f77 ⤸
```

Type in the program according to Figure 11-2. Don't forget to insert a tab (TAB key or CTRL-I) before each FORTRAN statement. The FORTRAN compiler is not case sensitive, so you can use either uppercase or lowercase. We've used lowercase for FORTRAN keywords and uppercase for variable names, but this is a matter of personal preference. AOS/VS FORTRAN 77 allows symbolic names of up to 32 characters, including letters, digits, the question mark (?), and the underscore (_); a symbolic name must begin with a letter or question mark.

```
C       This FORTRAN 77 program computes mortgage payments:
C               summary or full schedule.

        double precision  AMOUNT, RATE, MON_RATE, MON_PAYMENT
        double precision  BALANCE, INTEREST, INT_TO_DATE, PRINCIPAL
        character*10  ANSWER                    ! For user answers.

5       print *, "Enter amount, rate, years; e.g., 60000, .14, 25 [NL]<NL>"
        read *,  AMOUNT, RATE, IYEARS           ! Get numbers from console.

        print *, "Type F[ull] for full schedule, other char for summary. "
        read (*, '(A)')  ANSWER ! Formatted "A" read can get substring.

        MON_RATE = RATE/12.0    ! Compute the monthly interest rate.
        MONTHS = IYEARS*12      ! Compute the number of months.

C       Calculate monthly payment MON_PAYMENT and write to the console.
        MON_PAYMENT = AMOUNT*MON_RATE*(1.0+MON_RATE)**MONTHS/
     +                     ((1.0+MON_RATE)**(MONTHS)-1.0)

        print 110,  AMOUNT, RATE, IYEARS, MON_PAYMENT
110     format ("0", "Amount    = $", F9.2 ,/, " Interest Rate =",F7.4,/,
     +  " Loan Life is ",I3," Years",/, " Monthly Payment = $", F9.2, /)

C       Does user want full schedule?  If so, do it with a block IF.

        if ( (ANSWER(1:1) .eq. 'F') .or. (ANSWER(1:1) .eq. 'f') )  then

            BALANCE = AMOUNT    ! Balance starts as the original amount.
            INT_TO_DATE = 0.0   ! Interest To Date starts as 0.

C           To LISTFILE, write summary, header, and figures month by month.

            write (12,110)  AMOUNT, RATE, IYEARS, MON_PAYMENT

            write (12,120)
120         format (1X, " Num", 8X, "Interest", 6X, "Prin. Pay", 6X,
     +      "Prin. Bal.", 4X, "Interest Paid to Date", /)

            do 200  I = 1, MONTHS       ! DO for all the months.

C               Compute amount of interest in the monthly payment.
                INTEREST = MON_RATE * BALANCE
                PRINCIPAL = MON_PAYMENT - INTEREST ! Principal in payment.
                BALANCE = BALANCE - PRINCIPAL      ! Update loan balance.
C               Update the interest paid to date.
                INT_TO_DATE = INT_TO_DATE + INTEREST
C               Output the running totals and balances.
                write (12,140) I, INTEREST, PRINCIPAL, BALANCE, INT_TO_DATE
140             format (1X, I3, 7X, "$", F9.2, 5X, "$", F9.2, 5X, "$",
     +          F9.2, 8X, "$", F9.2, /)

200         continue
```

Figure 11-2. FORTRAN 77 MORTGAGE Program with Errors (continues)

```
      end if          ! End of full schedule in block IF.

      print *, "Type R to repeat, other char to stop. '            "
      read (*, '(A) )   ANSWER        ! Formatted "A" read.
      if ( (ANSWER(1:1) .eq. 'R') .or. (ANSWER(1:1) .eq. 'r') ) go to  5
      end
```

Figure 11-2. FORTRAN 77 MORTGAGE Program with Errors (concluded)

Compiling the FORTRAN 77 Program

Having created an initial version of MORTGAGE.F77, you are ready to compile it. Since the initial version of a program usually has some syntactical errors, use the /N switch, which directs the compiler not to produce an object module. Type

```
) f77/n mortgage.f77 ↓
```

The compiler displays its banner, with the date, time, revision number, and the switch option you included. Then it displays error messages:

```
Error 255 severity 2 beginning on line 57
Missing string delimiter in a character constant.

Error 255 severity 2 beginning on line 58
Missing string delimiter in a character constant.

Error 82 severity 3 beginning on line 58
Invalid syntax in this statement.  An END-OF-STATEMENT
was found where another token was expected.
```

F77 error messages are very specific. These identify lines 57 and 58. In line 57, a character constant starts with a quotation mark (") and ends with an apostrophe ('). In line 58, the format specification in the READ statement does not have a closing apostrophe.

Using the SED or SPEED text editor, correct the lines, producing the following changed lines:

```
print *, "Type R to repeat, other char to stop. "

read   (*, '(A)')  ANSWER      ! Formatted "A" read.
```

Leave the text editor, and recompile the program with the following command line:

```
) f77/lineid mortgage.f77 ↓
```

The /LINEID switch directs the F77 compiler to report the program line numbers involved if any runtime errors occur. This switch can help you identify logic errors in your sources, and is especially useful during program development whenever a program is likely to have runtime errors.

The compiler completes without error messages, and the CLI prompt returns. You can now link the program with the system Link utility.

Creating the Program File with Link

The Link command line for the MORTGAGE program is

```
) f77link mortgage.ob ↓
```

While Link builds the MORTGAGE program file, it displays the message,

```
LINK REVISION nnnn  ON  date  AT time
OPTIONS...
MORTGAGE.PR  CREATED
```

In FORTRAN, or any other high-level language, Link errors are rare. If you ever *do* receive a Link error message, and the text doesn't tell you how to fix the problem, see the *AOS/VS Link and Library File Editor (LFE) User's Manual.*

Executing the FORTRAN Program

To execute MORTGAGE.PR or any other program, simply type the XEQ command, and follow it with the program name and press NEW LINE.

```
) xeq mortgage ↓

Enter amount, rate, years; e.g., 60000, .14, 25 [NL]
```

Respond with some plausible figures such as a mortgage of $80,000 at 12.5% for 30 years; separate these figures with commas:

```
80000, .125, 30 ↓

Type F[ull] for full schedule, other char for summary.
```

Select a summary by entering any character, so press

↓

The MORTGAGE program then displays the loan statistics:

```
Amount = $80000.00
Interest Rate = .1250
Loan Life is 30 Years
Monthly Payment = $  853.81

Type R to repeat, other char to stop.
```

Looks good. Now repeat the program, type

```
r ↵
```

The MORTGAGE program prompts for the numbers again. Enter the same numbers, but request a full schedule this time:

```
80000, .125, 30 ↵

Type F[ull] for full schedule, other char for summary.    F ↵

Amount = $80000.00
Interest Rate = .1250
Loan Life is 30 Years
Monthly Payment = $  853.81

ERROR          21.
from line 34 of .MAIN.
FILE DOES NOT EXIST

        at location: n

        ERROR OCCURRED DURING ACCESS TO UNIT 12.

        TASK terminated
.
.
.
```

A fatal runtime error! And we find the CLI running on the terminal. The F77 message informs us that a file does not exist, and that the error condition occurred during access to Unit 12.

The source of the problem is in the preconnection to Unit 12. AOS/VS FORTRAN 77 has default preconnections between the following units and files:

Unit Number	File
9	@DATA
10	@OUTPUT
11	@INPUT
12	@LIST

The files @DATA, @OUTPUT, @INPUT, and @LIST are actually pointers to files; they're called *generic files*. As described in Chapter 3 (under the LISTFILE command), these file pointers aren't initially set to any filename; they exist for your convenience. You can set them to any legal filename

you want, but you *must* set them before using them. Thus, when the program tried to write to Unit 12 (on line 34), the system couldn't find the file to which @LIST points, so it aborted the program with a runtime error message.

To correct this, we can simply set @LIST to the terminal (filename @CONSOLE). This way, the system will display any data sent to @LIST on the terminal screen. Type

```
) listfile @console ⤵
```

Setting the list file to the terminal lets you see the output directly. Later, you'll direct output to a disk file.

Having set the @LIST file to the terminal, execute the program again. Then supply the same figures, and request a full schedule:

```
) xeq mortgage ⤵

Enter amount, rate, years; e.g., 60000, .14, 25 [NL]


80000, .125, 30 ⤵

Type F[ull] for full schedule, other char for summary.    F ⤵


Amount    = $ 80000.00
Interest Rate =  .1250
Loan Life is  30 Years
Monthly Payment = $   853.81

 Num        Interest      Prin. Pay      Prin. Bal.    Interest Paid to Date

  1       $   833.33     $    20.47     $ 79979.53        $   833.33

  2       $   833.12     $    20.69     $ 79958.84        $  1666.45
.
.
.
```

It works! Stop the program with CTRL-C, CTRL-B:

```
CTRL-C CTRL-B
*ABORT*
ERROR: CONSOLE INTERRUPT
ERROR: FROM PROGRAM
xeq mortgage
```

Now that you know it works, set @LIST to a disk file, and execute the MORTGAGE program again, providing the same figures:

```
) listfile mortgage.out ↓
) xeq mortgage ↓


Enter amount, rate, years; e.g., 6000, .14, 25 [NL]
80000, .125, 30 ↓

Type F[ull] for full schedule ...    f ↓

Amount     = $ 80000.00
Interest Rate =  .1250
Loan Life is  30 Years
Monthly Payment = $   853.81
```

You see the same summary figures as before, then wait a moment as MORTGAGE writes the full schedule to file MORTGAGE.OUT. Then the MORTGAGE program prompts for instructions. This time direct it to stop:

```
Type R to repeat, other char to stop.    s ↓
```

The MORTGAGE program terminates, closing the list file MORTGAGE.OUT. MORTGAGE.OUT will remain in the working directory FORTRAN. While it remains the list file, each full schedule you ask for when you run the MORTGAGE program will be appended to MORTGAGE.OUT. To print the file, type

```
) qpr mortgage.out ↓
QUEUED,  SEQ=n
```

Checking the printer, you'll find the full schedule, part of which is shown in Figure 11-3. A header page preceding the payment schedule describes your username, the output file pathname, and date. The entire schedule is eight or so pages long. The interest total at the end is pretty appalling.

```
Amount    = $ 80000.00
Interest Rate =  .1250
Loan Life is  30 Years
Monthly Payment = $   853.81

 Num        Interest       Prin. Pay       Prin. Bal.     Interest Paid to Date

  1       $   833.33     $    20.47     $ 79979.53        $   833.33

  2       $   833.12     $    20.69     $ 79958.84        $  1666.45

  3       $   832.90     $    20.90     $ 79937.94        $  2499.36

  4       $   832.69     $    21.12     $ 79916.82        $  3332.04

 10       $   831.33     $    22.47     $ 79785.40        $  8323.47
  .            .              .              .                .
  .            .              .              .                .
  .            .              .              .                .
359       $    17.51     $   836.29     $   845.00        $227361.43
360       $     8.80     $   845.00     $      .00        $227370.24
```

Figure 11-3. Part of the Full Payment Schedule from the MORTGAGE Program (FORTRAN 77)

Remember the List File

The list file is still set to MORTGAGE.OUT. You might want to reset it to @LIST with the CLI LISTFILE command. But the next time you log on, it will be set to the dummy filename @LIST, as when the MORTGAGE program first executed. So, if you later receive a *FILE DOES NOT EXIST* error message when you execute MORTGAGE and request a full schedule, just set the list file to @CONSOLE or a disk file.

You can see the flexibility of the list file preconnection to Unit 12. It allows you to set your output file at will from the CLI.

Using the SWAT® Debugger

Data General's SWAT® Debugger — if your system has it — can really ease the debugging phase of program development. To use SWAT, compile your program with the /DEBUG switch and request a listing with the /L=@LPT switch:

```
) f77/debug/l=@lpt mortgage.f77 ↵
```

Link the program using the /DEBUG switch:

```
) f77link/debug mortgage.ob ↵
```

Start up the program in SWAT:

```
) swat mortgage ↓
AOS/VS SWAT Rev...
>
```

In SWAT, you can set breakpoints by listing a line number with the BREAKPOINT command, or use the LIST command to display source lines and check program logic. You can start or run the program with the CONTINUE command, examine variables at breakpoints with the TYPE command, receive assistance at any point with the HELP command, and leave SWAT with the BYE command.

For example, a SWAT session with the MORTGAGE program might involve the following dialog:

```
> break  21 ↓
Set at :.MAIN:21

> con ↓
Enter amount, rate, years; e.g., 6000, .14, 25 [NL]

80000, .125, 30 ↓
Type F[ull for full schedule, other char for summary.      ↓

Breakpoint trap at :.MAIN:21.

> type mon_payment ↓
8.5380620987852...

> bye ↓
This Swat session is terminating.
```

With SWAT you can set breakpoints, as you did above on line 21, and examine program variables. By tracking the values in a program, you can determine where there's a breakdown in logic or in program execution.

What Next?

If you want to try experiment with another language or review earlier material, continue to the appropriate chapter. Or you can start writing your own F77 programs, using the *FORTRAN 77 Reference Manual.*

For details on FORTRAN 77 and SWAT manuals, as well as AOS/VS documentation in general, turn to Chapter 16.

End of Chapter

Chapter 12
Pascal Programming

This chapter shows you, in a hands-on session, how to develop and run a Pascal program under AOS/VS. If you're totally unfamiliar with the Pascal language, we recommend that you turn to a language reference, listed in Chapter 16. Once you have a sense of Pascal, return to this chapter for an example of AOS/VS program development.

Program Development Sequence

These are the steps you follow to create a program in AOS/VS Pascal:

1. Create or edit a Pascal source file with the SED or SPEED text editor:

   ```
   ) xeq sed filename.pas ↓
   ```

   or

   ```
   ) xeq speed/d filename.pas ↓
   ```

   With your chosen editor, type in the Pascal statements and comments that make up the program.

2. Compile the source file with the compile macro supplied by Data General:

   ```
   ) Pascal filename ↓
   ```

   The compile command line can include the /DEBUG, /LINEID, /N, and other switches described in the CLI Help messages (`HELP *PASCAL`).

3. If there are any errors from the compiler, return to step 1 and fix the offending statement(s). If there are no compile errors, go to step 4.

4. Link the object file to produce an executable program:

   ```
   ) paslink filename ↓
   ```

   If you plan to debug the program, you would include the /DEBUG switch on both the link and the compile commands.

5. Execute the program with the CLI command,

   ```
   ) xeq filename ↓
   ```

6. If the program runs as you want it to, go to step 9.

7. Identify logic errors using runtime error messages or erroneous output. Or, if you included the /DEBUG switch and your system has the SWAT interactive debugger, debug the program with the command,

   ```
   ) swat filename ↓
   ```

8. Return to step 1 and fix any erroneous statement(s).

9. Rebuild the program for maximum efficiency. Compile again with the /OPTIMIZE switch,

   ```
   ) Pascal/optimize filename ↓
   ```

10. And return to step 1 to insert any necessary code.

11. You're done! You can go on to write your own programs under AOS/VS.

This chapter guides you through all the steps you need to develop and execute a Pascal program.

The Pascal Program Example

The sample program calculates home mortgage payments, and produces a schedule of monthly principal and interest. The program uses only ordinary arithmetic operations, calls no subroutines, and is less than a page long. The program uses two formulas. You don't need to understand how these formulas work to use the program as an example.

The mortgage formulas are those used by U.S. banks. Different formulas are used outside the United States. If you live outside the United States, treat the formula parts of the program as examples only. (Later, you might want to replace them with ones appropriate to your nation.)

Figure 12-1 shows a flow chart for this program. Figure 12-2 shows the initial version of the program MORTGAGE.PAS itself, complete with errors.

Even if you decide not to copy this program, you should at least examine Figures 12-1 and 12-2 before reading the rest of the chapter.

To compile, link, and run Pascal programs, your system must have the appropriate compiler files and libraries. The directory that holds these, usually directory :UTIL:PASCAL, must be on your search list. For example, to set your search list when the Pascal directory is subordinate to :UTIL, you would type

```
) sea :util:Pascal [!sea] )
```

The directory :UTIL:PASCAL is now on your search list until you change it or log off the system.

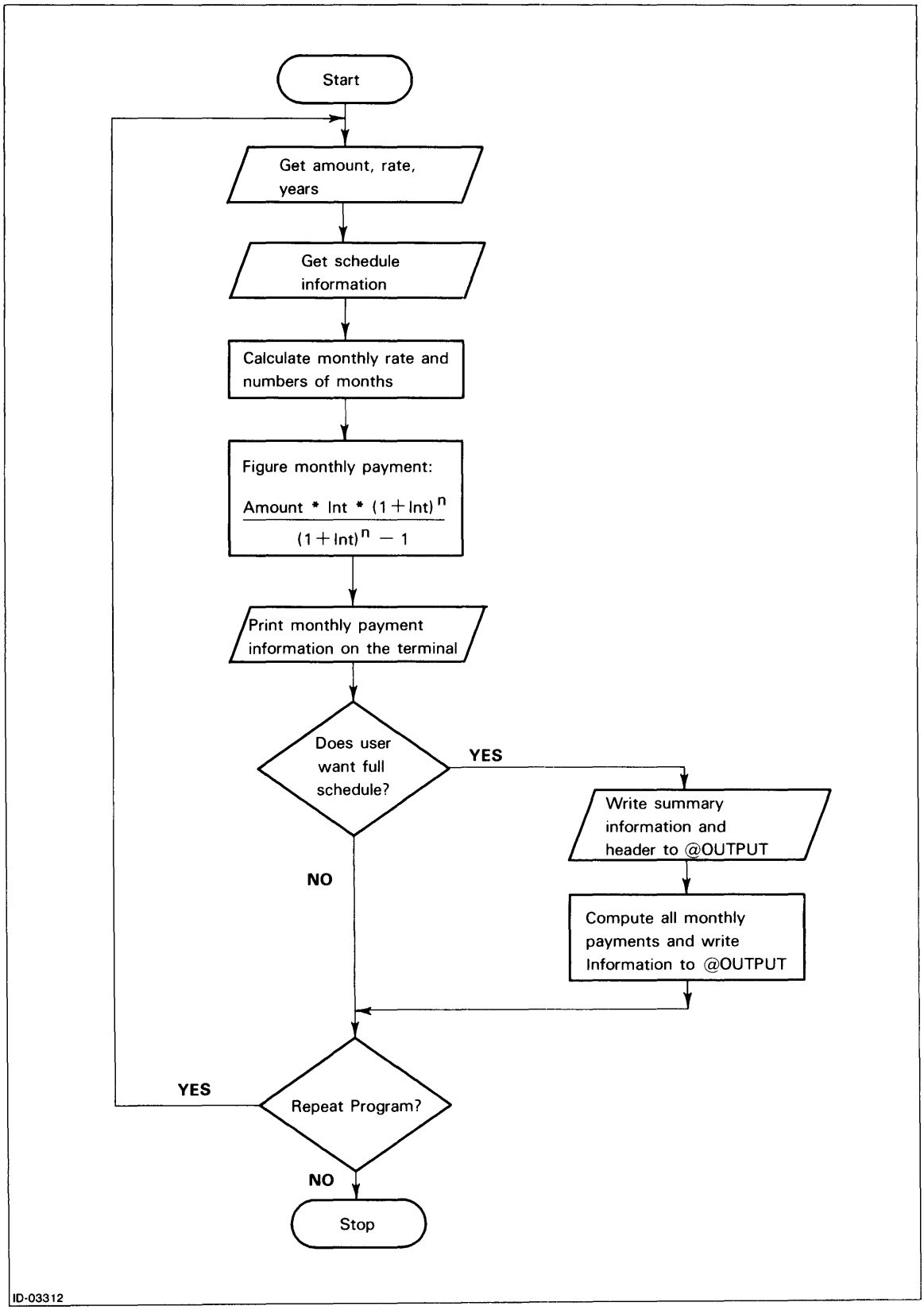


Figure 12-1. MORTGAGE Program Flow Chart (Pascal)

Writing the Pascal Program

If you want to copy the MORTGAGE program, we suggest that you first create a directory for it. This will prevent conflicts with other programs that have the same name and will encourage you to place all your Pascal programs in the same place. For example, type

```
) dir/i ⤸
) cre/dir Pascal ⤸
) dir Pascal ⤸
```

Now, execute the SED or SPEED text editor, and create the source file. We suggest that the source filename end with the conventional Pascal extension, .PAS.

```
) xeq sed mortgage.pas ⤸
```

or

```
) xeq speed/d mortgage.pas ⤸
```

Type in the program as it appears in Figure 12-2. Although Pascal is not space sensitive, it is a good idea to use a block format, such as the one in Figure 12.2, because it makes the program easy to read and maintain. Notice that program comments are set off with braces ({ }). On older terminals, which don't have the brace key, use the left-parenthesis–asterisk pair, (*, to open a comment, and the asterisk–right-parenthesis pair, *), to close it. In Pascal, semicolons separate one statement from the next, and the program ends with a period.

Pascal allows symbolic names of 32 characters, consisting of letters, digits, the underscore (_), the question mark (?), and the dollar sign ($).

```
{ This Pascal program computes mortgage payments: summary or full schedule. }
program MORTGAGE(INPUT,OUTPUT);

var
    AMOUNT, RATE, R, PAY:       double_real; { For today's rates. }
    BAL, INTEREST, ITD, PRIN:   double_real;
    ANSWER:                     char; { For user answers. }
    YEARS, MONTHS, MM, I:       integer;

begin
    repeat
        write('Enter amount, e.g., 60000 [NL]  $ '); readln(AMOUNT);
        write('Enter rate,   e.g. 14 [NL] ');     readln(RATE);
        write('Enter years, e.g., 25 [NL] ');     readln(YEARS);

        R:= RATE/12;            { Change yearly RATE to monthly rate R. }
        MONTHS:= YEARS*12;              { Change years YEARS to months MONTHS. }

        write('Type F for full schedule, other char for summary. ');
        readln(ANSWER);

{ Calculate monthly payment, PAY, and write to the console. }

        PAY:= AMOUNT*R*power(1+R,MONTHS)/ (power(1+R,MONTHS)-1);

        writeln('Amount        = $', AMOUNT:9:2);
        writeln('Interest Rate =', RATE:7:4);
        writeln('Loan Life is ',YEARS:3,' Years');
        writeln('Monthly Payment = $', PAY:9:2);

{ Did user want full schedule? }

        if ((ANSWER= 'F') or (ANSWER= 'f')) then begin
            BAL:= AMOUNT;       { Balance starts as the original amount.}
            MM:= MONTHS;        { Save original number of months in MM. }
            ITD:= 0;            { Interest To Date starts as 0. }

{ Write summary, header, and figures month by month. }

            writeln('  Num        Interest      Prin. Pay',
                        '      Prin. Bal.    Interest Paid to Date');

            for I:=1 to MM do begin     { DO for all the months.. }
                PRIN:= BAL*R/ (power(R+1,MONTHS)-1);{ Amt of prin. in payment.}
                MONTHS:= MONTHS-1;              { Decrement month. }
                INTEREST:= PAY-PRIN;    { Get amt of interest in PAYment. }
                BAL:= BAL-PRIN;         { Update loan balance. }
                ITD:= ITD+INTEREST;     { Update interest paid to date. }
                writeln(' ',I:3,'       $',INTEREST:9:2,'     $',PRIN:9:2,
                  '     $',BAL:9:2,'        $',ITD:9:2) { Write figures. }
                end

            end;                { End of full sched. }

        write("Type R to repeat, other char to stop. ');
        readln(ANSWER);
        until ((ANSWER< > 'R') and (ANSWER< > 'r'))
    end.
```

Figure 12-2. Pascal MORTGAGE Program with Errors

Compiling the Pascal Program

Having created an initial version of MORTGAGE.PAS, you are ready to compile it. Since the initial version of a program usually has some syntactical errors, use the /N switch, which directs the compiler not to produce an object module.

```
) Pascal/n  mortgage.pas ↓
```

The compiler displays its banner, showing the date, time, revision number, and any switch option you selected. Then it displays error messages:

```
Error 310 severity 3 beginning on line 55  (Line 55 of file MORTGAGE.PAS)
Unknown Pascal character.
        write("Type R to repeat, other char to stop.  ');
              ^

Error 300 severity 3 beginning on line 55  (Line 55 of file MORTGAGE.PAS)
Syntax error.
        write("Type R to repeat, other char to stop.  ');
              ^
.
.
.
```

Pascal error messages are very specific. They identify the type of error, the line where it occurred, and the file in which the code is located. In this case, the error message prints a caret to mark the position where the syntax broke down. An error exists in line 55, where the write statement contains an illegal character: the quotation mark should be an apostrophe ('). Using the SED or SPEED text editor, correct the line, producing the following line:

```
        write('Type R to repeat, other char to stop. ');
```

After leaving the editor, compile the program again. This time use both the /LINEID and /DEBUG switches. The /LINEID switch tells Pascal to report the program line numbers involved if any runtime errors occur. The switch can help you identify logic errors in your sources; it is useful during program development whenever a program might have runtime errors. Type

```
) Pascal/lineid/debug  mortgage.pas ↓
```

The compiler completes without error messages, and the CLI prompt returns. You can now link the program with the system Link utility.

Creating the Program File with Link

The Link command line for the MORTGAGE program is

```
) paslink/debug  mortgage.ob ↓
```

While Link builds the MORTGAGE program file, it displays the message,

```
LINK REVISION nnnn  ON  date  AT  time
OPTIONS: LINK/DEBUG
MORTGAGE.PR CREATED
```

In Pascal, or any other high-level language, Link errors are rare. If you ever *do* receive a Link error message, and the text doesn't tell you how to fix the problem, see the *AOS/VS Link and Library File Editor (LFE) User's Manual.*

Executing the Pascal Program

To execute MORTGAGE.PR or any other program, simply type the XEQ command, followed by the program name and press NEW LINE.

```
) xeq mortgage ↵
```

The program starts up, and prompts for the amount, rate, and terms of the mortgage. Respond with some plausible figures, such as a sum of $80,000 at 12.5% for 30 years:

```
Enter amount, e.g. 60000  $   80000 ↵
Enter rate, e.g. 14           12.5 ↵
Enter years, e.g. 25          30 ↵
```

Then the program queries you for the type of report:

```
Type F for full schedule, other char for summary.
```

Press NEW LINE for a summary:

```
↵
```

But nothing happens! Press the NEW LINE key again. The system comes alive and displays the following message:

```
ERROR         71175.
from line 24 of MORTGAGE.

Call Traceback:

from fp=16000006750,  pc=0.SERROR+15
from fp=16000006716,  pc=DEF?ON+71
from fp=16000006604,  pc=MORTGAGE+250,  line 24 of MORTGAGE
.
.
.
Floating point overflow.
```

A floating point overflow occurred on line 24 of the program. Somewhere in its calculations, the program tried to work with a number greater than it could handle, so it stopped. To investigate the problem, you need to run MORTGAGE.PR in the debugger.

Using the SWAT Debugger

Data General's SWAT Debugger — if your system has it — can really ease the debugging phase of program development. To use SWAT, you compile and link your program with the /DEBUG switch, just as you did. (It's also a good idea to request a compiler listing with the /L switch.) For example, when you compiled the mortgage program the second time, you used the command line

```
) Pascal/debug/lineid  mortgage.pas ↓
```

and linked the program, using the /DEBUG switch,

```
) paslink/debug mortgage.ob ↓
```

Because you did so, you can now start the program in the SWAT debugger and see what the problem is on line 24.

```
) swat  mortgage ↓
AOS/VS SWAT Rev nnnn ON date AT time
PROGRAM  :UDD:ALEXIS:MORTGAGE
```

In SWAT you can set stopping points in the program with the BREAKPOINT command, or check logic by displaying source lines with the LIST command. To start, or restart the program, use the CONTINUE command. Exit from the debugger with the BYE command. Other commands include the following: the TYPE command, which allows you to examine variables at breakpoints; the DESCRIBE command, which provides information about program variables; and the HELP command, which supplies information about the debugger in general.

To trace the floating point problem, set a breakpoint at line 24, so you can examine the program values at that time. The program will run to line 24 and stop.

```
> break  24 ↓
Set at :MORTGAGE:24
```

To run the program in SWAT, use the CONTINUE command, and then respond to the program prompts. For example,

```
> con ↓
Enter amount, e.g. 60000   $  80000 ↓
Enter rate, e.g. 14           12.5 ↓
Enter years, e.g. 25          30 ↓
Type F for full schedule, other char for summary.
```

This time, we select a full schedule by typing

```
f ↓

Breakpoint trap at :MORTGAGE:24
```

The program stopped at line 24. Let's look at the line now, using the LIST command. Type

```
> list 24 ⤸
24C             PAY:=AMOUNT*R*power(1+R,MONTHS)/ (power(1+R,MONTHS)-1);
```

Line 24 is where the program calculates the monthly payment. Now check the program variables in the line to see where things are going wrong.

```
> ty amount ⤸
8.000000000000000E+04

> ty r ⤸
1.041666666666667E+00

> ty months ⤸
360
```

If you plug these values into the formula on line 24, you have, for the numerator, the statement 80,000 * 1.04 * 2.04^{360}. Here's the problem: 2.04 raised to the 360th power would cause a floating point overflow. Why are the numbers so large? To see the source of the problem, return to the definition phase of the program to check the variables. Type

```
> list @all ⤸


{ This Pascal program computes mortgage payments: summary or full schedule. }
program MORTGAGE(INPUT,OUTPUT);

var
    AMOUNT, RATE, R, PAY:       double_real; { For today's rates. }
    BAL, INTEREST, ITD, PRIN:   double_real;
    ANSWER:                     char; { For user answers. }
    YEARS, MONTHS, MM, I:       integer;
.
.
.
        R:= RATE/12;            { Change yearly RATE to monthly rate R. }
```

The problem arises in the statement R:=RATE/12. You entered an interest rate of 12.5, and the program should convert that to a decimal, .125. But it doesn't, thus the overflow. To correct the problem, you add /100 to the expression on line 24. The new line will read

```
        R:= RATE/12/100;        { Change yearly RATE to monthly rate R. }
```

Now that you've solved the floating point problem, exit from SWAT by typing

```
> bye ⤸
This Swat session is terminating.
```

Final Version of the MORTGAGE Program

Before you edit the source file, what about the fact that nothing happened when we responded with a single NEW LINE to the question,

```
Type F for full schedule, other char for summary.   ↓
```

The program froze until we entered a second NEW LINE. Why? Let's look at the code:

```
19      write('Type F for full schedule, other char for summary. ');
20      readln(ANSWER);
```

The READLN(ANSWER) statement on line 20 handles your response to the question on line 19. How does READLN work? By design, READLN looks for two or more characters: it assigns the first character(s) to the variable ANSWER, and then looks for a delimiter, in this case NEW LINE. Because the READLN function needs at least two values, it didn't respond to our first NEW LINE. Instead it assigned NEW LINE to ANSWER and waited for a delimiter, which we eventually entered.

However the program should handle the case of a single NEW LINE. You need to modify line 20 to handle a single character, but what will do the job?

The EOLN function reads a single character. It will read the input and test for a NEW LINE. If the character is a NEW LINE, it returns TRUE. But when the character is not NEW LINE, it returns FALSE.

In this way, if the first character entered is a NEW LINE, it can set ANSWER to space and go on; if more than one character is entered, the program can proceed to the READLN statement.

You can add the EOLN function to line 20, and also in line 56, where READLN occurs again.

Execute the SED or SPEED text editor, and correct the lines 20, 24, and 56 so the program reads

```
20      if eoln(INPUT) then ANSWER:= ' ' else readln(ANSWER); reset(INPUT);

24      R:= RATE/12/100;        { Change yearly RATE to monthly rate R. }

56      if eoln(INPUT) then ANSWER:= ' ' else readln(ANSWER); reset(INPUT);
```

Now try the program. Compile and link it one last time:

```
) Pascal mortgage.pas ↓

) paslink mortgage.ob ↓
```

Then try it out.

```
) xeq mortgage ↓


Enter amount, e.g. 60000   $ 80000 ↓
Enter rate, e.g. 14          12.5 ↓
Enter years, e.g. 25         30 ↓

Type F for full schedule, other char for summary.
```

Select a summary to test the new expression in line 20: Press

```
↵
```

The MORTGAGE program displays a summary, and waits for directives.

```
Amount = $80000.00
Interest Rate = 12.5000
Loan Life is 30 Years
Monthly Payment = $  853.81

Type R to repeat, other char to stop.
```

That much works! Now let's run it one more time and request a full schedule. Type

```
r ↵

Enter amount, e.g. 60000    $  80000 ↵
Enter rate, e.g. 14            12.5 ↵
Enter years, e.g. 25           30 ↵

Type F for full schedule, other char for summary.    f ↵

Amount = $80000.00
Interest Rate =12.500
Loan Life is 30 Years
Monthly Payment = $  853.81
   Num      Interest      Prin.Pay     Prin. Bal.     Interest Paid to Date
    1    $   833.33    $   20.47   $ 79979.53         $   833.33
    2    $   833.12    $   20.69   $ 79958.84         $  1666.45
    3    $   832.90    $   20.90   $ 79937.94         $  2499.36
    4    $   832.69    $   21.12   $ 79916.82         $  3332.04
    5    $   832.47    $   21.34   $ 79895.48         $  4164.51
.
.
.
```

The full schedule also works, so stop the program with CTRL-C, CTRL-B:

```
CTRL-C  CTRL-B

*ABORT*
CONSOLE INTERRUPT
ERROR: FROM PROGRAM
xeq,mortgage
```

Now that the MORTGAGE program runs, you can use the Pascal Optimizer to generate the most efficient code possible.

Building the Program for Maximum Efficiency

The Pascal compiler does runtime error checking by default; therefore there's more to building your program for maximum efficiency than simply optimizing. Runtime error checking means that the compiler, without being requested, builds extra code into the MORTGAGE program to evaluate expressions and test the validity of input. Sometimes you'll want runtime error checking, and other times you won't. You can determine where it's necessary, once the compiler reports on statements that generate the extra code.

To see how the Optimizer works, compile the program again. This time use the /OPTIMIZE switch. Type

```
) Pascal/optimize mortgage.pas ↵
```

The Pascal compiler displays its banner, then a warning:

```
%Warning 449 Severity 1 beginning on line 20 (Line 20 of file MORTGAGE.PAS).
The generated code will be less than optimal when /OPTIMIZE
switch is used in conjunction with ... any directive setting
which enables runtime checking.
```

In this case, the compiler warns us that it's built in code to test the value of ANSWER in line 20:

```
19      write('Type F for full schedule, other char for summary. ');
20      if eoln(INPUT) then ANSWER:= ' ' else readln(ANSWER); reset(INPUT);
```

In this case, we don't care about the value of ANSWER because there is nothing you can supply that will make the program execute incorrectly. So we can generate a more efficient program by eliminating range checking here. Then the MORTGAGE program will run faster and use less address space.

To compile without range checking, execute the compiler with the /OVERRIDE=R- switch,

```
) Pascal/optimize/override=r-  mortgage.pas ↵
```

This time the program compiles without any warnings, and you can be sure you have the most efficient program possible.

Printing the Full Schedule

If you would like a printed copy of the full schedule, you can run the MORTGAGE program in batch. This way, any output directed to @CONSOLE is redirected to the line printer. To print the schedule, first create a command file in either SED or SPEED; for example, type

```
) xeq sed print.cli ↵
```

or

```
) xeq speed/d print.cli ↵
```

Once in the editor, enter the following seven lines of text into the file PRINT.CLI:

```
xeq/m mortgage.pr
80000
12.5
30
f
!       (just press NEW LINE)
)
```

Note the /M switch on the CLI command XEQ. It tells the CLI that the file contains arguments to the MORTGAGE program. Therefore, the CLI should accept input from the command file rather the keyboard. The next five lines are familiar by now: they represent input to the MORTGAGE program. The final line, the right parenthesis, notifies the CLI that the command file is complete.

You can execute the print macro from your terminal, to obtain a screen display. Or, to obtain a hard copy of the full schedule, execute the print macro in batch with the QBATCH command,

```
) qbatch print !
QUEUED,  \ SEQ=n
```

Checking the printer, you'll find a printed header that describes your logon message, default ACL, username, and so on. Following the header you'll find the full schedule, part of which is shown in Figure 12-3. The entire schedule is eight or so pages long. The interest total at the end (which the Figure 12-3 doesn't show but your listing will) is pretty appalling.

```
Amount = $80000.00
Interest Rate =12.500
Loan Life is 30 Years
Monthly Payment = $  853.81
   Num      Interest      Prin.Pay     Prin. Bal.     Interest Paid to Date
    1    $   833.33    $   20.47   $ 79979.53        $   833.33
    2    $   833.12    $   20.69   $ 79958.84        $  1666.45
    3    $   832.90    $   20.90   $ 79937.94        $  2499.36
    4    $   832.69    $   21.12   $ 79916.82        $  3332.04
    5    $   832.47    $   21.34   $ 79895.48        $  4164.51
    6    $   832.24    $   21.56   $ 79873.92        $  4996.76
    7    $   832.02    $   21.79   $ 79852.13        $  5828.78
    8    $   831.79    $   22.01   $ 79830.12        $  6660.57
    9    $   831.56    $   22.24   $ 79807.88        $  7492.13
   10    $   831.33    $   22.47   $ 79785.40        $  8323.47
   11    $   831.10    $   22.71   $ 79762.69        $  9154.56
   12    $   830.86    $   22.94   $ 79739.75        $  9985.42
    .
    .
    .
```

Figure 12-3. Beginning of the Full Payment Schedule from the MORTGAGE Program (Pascal)

What Next?

If you want to experiment with another language, or review material on the CLI or a text editor, turn to the appropriate chapter.

For details on Pascal and SWAT documentation, as well as AOS/VS documentation in general, turn to Chapter 16.

End of Chapter

Chapter 13
Assembly Language Programming

This chapter can't teach you all about programming in AOS/VS assembly language, but it will introduce some of the basic procedures. We assume that you have some familiarity with the concept of assembly language. It will help if you have some awareness of the ECLIPSE MV/Family instruction set. All MV/Family instructions are described in the processor-specific manuals listed in Chapter 16, Documenation Guide. We describe some of these instructions in this chapter.

If you have programmed in 16-bit AOS assembly language, this experience will help because of the similarity between the ECLIPSE and the MV/Family architecture and the AOS and AOS/VS operating systems. The MV/Family instruction set provides for 16-bit ECLIPSE instructions: you can write 16-bit AOS assembly language programs, using the 16-bit instruction set, assemble with the AOS/VS Macroassembler, and execute them on AOS/VS just as you would on AOS. If you want 100% compatibility, then you can assemble 16-bit programs under AOS/VS with the 16-bit assembler (MASM16), link with the /SYS=VS16 switch, then execute (XEQ) the programs on AOS/VS or AOS.

However, we assume — in this chapter and Chapter 14 — that you want to take advantage of the full 32-bit power of the MV/Family computers. The machine instructions that allow this differ somewhat from the 16-bit machine instructions.

Even if you are not familiar with ECLIPSE computers or AOS assembly language, you will profit by reading this chapter and the next one. Assembly language is a picture of system fundamentals; an understanding of it is essential if you care about bedrock software or hardware operations. To really understand the assembly language program described in the next chapter — you will probably need some of the basics described in this chapter.

Program Development

FORTRAN, COBOL, and BASIC programs consist of statements that a compiler (or interpreter) translates into machine-level instructions. So do assembly language programs. But the program that does the translation is called an assembler — specifically the AOS/VS Macroassembler (MASM) utility.

These are the steps you follow to develop an assembly language program:

1. Create or edit an assembly language source file (sometimes called a module) with the SED or SPEED text editor:

   ```
   ) xeq sed pathname ↓
   ```

   or

   ```
   ) xeq speed/d pathname ↓
   ```

   With your chosen editor, type in the assembly language statements that make up the program.

2. Assemble the source file with the CLI command,

   ```
   ) xeq masm/l=@lpt pathname ↓
   ```

   The /L=@LPT switch sends the output to a line printer, but you can also send it to a disk file with the /L=filename switch.

3. If there are assembly errors, return to step 1 and fix them. If there are no errors, go to step 4.
4. Link the object file to produce an executable program:

   ```
   ) xeq link pathname ↓
   ```

   If you receive any link errors, return to step 1.
5. Execute the program with the CLI command,

   ```
   ) xeq pathname optional-arguments ↓
   ```
6. If the program runs as you want it to, go to step 9.
7. Try to identify any problem, using runtime error messages or erroneous output. If you find errors, return to step 1 and fix the problems.
8. Debug the program with the command,

   ```
   ) debug pathname optional-arguments ↓
   ```

   The AOS/VS Debugger is a utility that allows you to stop the program, examine memory locations, and make temporary changes. Having used the Debugger to find problems, go to step 1 and fix them.
9. You're done! You can write original programs or continue to another topic of interest.

You will probably have to repeat the process of assembling and linking, then executing and debugging several times before you produce a program that works as you want it to.

About MASM, the Macroassembler

MASM is a utility program supplied with all AOS/VS systems. (The Macroassembler derives its name from instruction sequences called macros. After defining a sequence of instructions as a macro, you use the macro name instead of all the instructions it contains.) MASM can assemble 16- and 32-bit programs. However, a 16-bit version called MASM16 is also supplied to create 16-bit programs that are compatible with AOS MASM.

Typically, you execute MASM with the command:

```
xeq masm/l=@lpt pathname
```

where the /l=@lpt switch directs MASM to generate a program listing and send it to the line printer, and pathname gives the name of an assembly language source file.

The assembly listing shows locations, assembled values, your original code, and errors, if any. Unless there are serious errors, MASM also produces an object module, called pathname.OB.

Assembly errors are analogous to FORTRAN or COBOL compiler errors. They usually indicate syntactical errors, not errors in structure or logic.

Source Code and Assembly Listings

Each assembly language statement is a line of text. This text can include assembler pseudo-operatives, called pseudo-ops, which direct the assembly process but do not result in any final program instructions themselves. Pseudo-ops are roughly equivalent to nonexecutable statements in FORTRAN.

Each line of source code that you write will have one or more of the following components:

- A *label* — a name for a location that must end with a colon.
- A *directive* — an instruction, a system call, or a pseudo-op.
- *Arguments*, separated by spaces or commas.
- A *comment*, preceded by a semicolon.

You can write any or all of these *in the order given.* Column position isn't important, but the order is. The sequence must be

label: directive argument(s) ; comment

Each label must end with a colon. Spaces, tabs, and — at most, one comma must separate the directive and argument(s), if any. You may or may not have tabs or spaces between a label and a directive, and an argument and a comment.

Each comment begins with a semicolon and extends to the end of the line. MASM does not process comments; it merely passes them on intact to the assembly listing. Figure 13-1 shows you a page from an assembly listing.

```
SOURCE: WRITE            MASM 06.00.00.00       04-OCT-85 12:36:26  PAGE   1


01                                  ;  This program gets a filename from the CLI XEQ
02                                  ; command, opens the terminal and the specified file,
03                                  ; writes a prompt to and reads a line from the
04                                  ; terminal, and writes the line to terminal and file.
05                                  ; To stop it, type NO<NEW LINE>.
06
07                                          .TITLE  WRITE
08                                          .ENT    WRITE     ; For debugging
09                                          .NREL             ; Unshared partition.
10
11                                  ; Get filename; open terminal and specified file.
12
13                                  WRITE:  ?GTMES  CLIMS     ; Get the filename.
14 000006 UC 115270                         WBR     ERROR     ;    Error, process it.
15                                          ?OPEN   FILE      ; Open file, create if needed.
16 000015 UC 114370                         WBR     ERROR     ;    Error, process it.
17                                          ?OPEN   CON       ; Open console.
18 000024 UC 111470                         WBR     ERROR     ;    Error, process it.
19
20                                  ; Write prompt, read line, check for NO, write files.
21
22 000025 UC 122071 000144          LOOP:   XLEFB   0, PROMPT*2  ; Get bptr to prompt.
23 000027 UC 121431 000136                  XWSTA   0, CON+?IBAD  ; Put in CONsole pkt.
24                                          ?WRITE  CON       ; Write prompt to CONsole.
25 000037 UC 110170                         WBR     ERROR     ;    Error, process it.
26 000040 UC 122071 000354                  XLEFB   0, BUFF*2 ; Get bptr to I/O buffer.
27 000042 UC 121431 000123                  XWSTA   0, CON+?IBAD  ; Put in CONsole pkt.
28                                          ?READ   CON       ; Read a line from CONsole.
29 000052 UC 104670                         WBR     ERROR     ;    Error, process it.
30 000053 UC 143051 047117                  NLDAI   'NO',  0  ; Put NO term. in AC0.
31 000055 UC 125451 000151                  XNLDA   1, BUFF   ; Get first WORD (2
32                                                            ;  bytes) of buffer.
33 000057 UC 120611                         WSNE    1, 0      ; Skip if Not Equal NO.
34 000060 UC 104370                         WBR     GOODBYE   ;    User typed NO, exit.
35                                          ?WRITE  CON       ; Not NO. Echo line,go on.
36 000067 UC 101170                         WBR     ERROR     ;    Error, process.
37                                          ?WRITE  FILE      ; Write line to FILE.
38 000076 UC 100270                         WBR     ERROR     ;    Error, process.
39 000077 UC 164670                         WBR     LOOP      ; Do it all again.
40
41                                  ;  Process error and/or return here.
42
43 000100 UC 153051 150000          ERROR:  NLDAI   ?RFEC+?RFCF+?RFER, 2 ; Get err flgs.
44 000102 UC 100270                         WBR     BYE       ; Skip subtraction.
45 000103 UC 150531                 GOODBYE: WSUB   2, 2      ; Set for good return.
```

```
43 000100 UC 153051 150000          ERROR:  NLDAI   ?RFEC+?RFCF+?RFER, 2
44 000102 UC 100270                         WBR     BYE
45 000103 UC 150531                 GOODBYE: WSUB   2, 2
```

Line number

Location counter

Location counter relocation symbol

Data field for first word (and third word, if any)

Data field for second and fourth words (if any)

Source line

DG-27431

Figure 13-1. Example of an Assembly Language Listing

Note, in Figure 13-1, that MASM created five columns of numbers and letters. To the right of these is the source code that the user entered to MASM.

In the source program, you'll usually start each line of code in column 1 or tab in to column 8. But in the assembly listing, the source code usually starts in column 32 and extends to the end of the line. The enlarged view of the listing shows some source lines that use each source component. For example, look at the last line:

Label	Directive	Argument(s)	Comment
GOODBYE:	WSUB	2, 2	; Set for

The label is at the left, the directive next, arguments next, and the comment is on the right. The aligned labels, directives, arguments, and comments in the source code makes the listing easier to read.

Figure 13-1 also shows you what the columns mean. On each page the leftmost column contains line numbers. The assembler usually starts a new listing page after the sixtieth line. At the top right of each page is the page number. MASM includes both page and line numbers in the alphabetical cross-reference it produces with each program listing.

Next to the line number column is the column that shows the relative location of each line of code in the final program. The numbers are octal and represent the location counter. Each assembled word adds 1 to this counter.

Next to the location counter is the relocation symbol for the location counter, which contains two letters — in this case UC, meaning Unshared Code. This is the relocation type. If the program had specified shared code (with the .NREL pseudo-op, described later), the column would contain the letters SC.

To the right of the relocation symbol for each instruction, there is another octal column. This is the value of the first 16-bit word assembled from the instruction. If the instruction assembles into one 16-bit word, this is the whole value. If MASM cannot know this value, perhaps because it refers to another module, the listing shows the contents as 000000; or if the instruction is .WORD A+5, where A is defined externally, it will show 000005 on the listing.

If an instruction requires two words, there is another octal number in the data field for the second word. You can see that instruction WBR assembles into one word, whereas XLEFB (location 25) assembles into two words. You can also see how the location counter moved from 000025 to 000027 to reflect the storage needs of XLEFB.

Again from Figure 13-1, you'll note that certain directives receive nothing but a line number. Those that begin with a period, such as .TITLE, are assembler pseudo-ops that are not included in the final program. The directives that start with ?, such as ?GTMES, are system calls to AOS/VS. They are macros and the assembler doesn't show their expansions. You can see that they take some space, though — the ?GTMES call in line number 13 consumes locations 000000 through 000005.

After creating the listing, MASM always creates an alphabetical cross-reference with user symbols and system calls. This cross-reference can help you find routines in big programs and can help you spot typing errors when you're debugging. Figure 13-2 shows you a sample page from a cross-reference listing.

The cross-reference list describes the page and line number where each user symbol, such as a label, is defined, and the page and line number of each reference to this symbol. The letters MA mean MACRO, the number sign (#) indicates the place where the symbol was defined, and EN means ENTRY, described in this chapter's section on Pseudo-Ops.

```
XREF:   WRITE            MASM 06.00.00.00       04-OCT-85 13:03:58  PAGE    3

?APND     00000000200      2/38
?GARG     00000000003      2/04
?GNUM     00000000001      2/05
?GREQ     00000000000      2/03
?GRES     00000000004      2/07
?GTLN     00000000006      2/02    2/09
?GTMES    00000000000 MA   1/13
?IBAD     00000000004      1/23    1/27    2/18    2/40
?IBLT     00000000030      2/15    2/28    2/36    2/54
?ICRF     00000040000      2/17    2/38
?IDEL     00000000016      2/26    2/52
?IFNP     00000000014      2/22    2/47
?IMRS     00000000003      2/24    2/50
?IRCL     00000000007      2/20    2/43
?IRLR     00000000010      2/45
?ISTI     00000000001      2/16    2/37
?OFCE     00000000040      2/38
?OFIO     00000000030      2/17    2/38
?OPEN     00000000000 MA   1/15    1/17
?READ     00000000000 MA   1/28
?RETURN   00000000000 MA   1/47
?RFCF     00000100000      1/42
?RFEC     00000010000      1/42
?RFER     00000040000      1/42
?RTDS     00000000002      2/17    2/38
?SYST     00000000000 MA   1/14    1/16    1/18    1/25    1/29    1/36    1/38
                           1/48
?WRITE    00000000000 MA   1/24    1/35    1/37
?XCALL    00000000001      1/14    1/14    1/16    1/16    1/18    1/18    1/25
                           1/25    1/29    1/29    1/36    1/36    1/38    1/38
                           1/48    1/48
BUFF      00000000251      1/26    1/31    2/19    2/41    2/57#
BYE       00000000105      1/44    1/47#
CLIMS     00000000116      1/14    2/02#   2/03    2/05    2/07    2/09
CNAME     00000000214      2/23    2/30#
CON       00000000164      1/18    1/23    1/25    1/27    1/29    1/36    2/15#
                           2/16    2/18    2/20    2/22    2/24    2/26    2/28
ERROR     00000000100      1/14    1/16    1/18    1/25    1/29    1/36    1/38
                           1/42#   1/48
FILE      00000000221      1/16    1/38    2/36#   2/37    2/40    2/43    2/45
                           2/47    2/50    2/52    2/54
GOODBYE   00000000104      1/34    1/46#
LOOP      00000000025      1/22#   1/39
NAME      00000000124      2/08    2/11#   2/48
PROMPT    00000000111      1/22    1/52#
WRITE     00000000000 EN   1/08    1/13#   2/59
```

Figure 13-2. An Assembly Language Cross-Reference Listing

Symbols

Practically every line you write in assembly language will include a symbol, because each machine language instruction is a symbol, such as WBR for "wide branch to." Other symbols are assembler pseudo-ops, system calls, and their related locations.

In addition, you will create your own symbols. You will want to name memory locations (e.g., START for the beginning of your program or LOOP for the beginning of a loop), so that you can refer to them symbolically. Any symbols you create must be unique within a program module and must adhere to the following format:

first character *[succeeding character(s)]* break

where

first character	is any letter from A–Z, a–z, period (.), question mark (?), or dollar sign ($).
succeeding character(s)	are A–Z, a–z, 0–9, period (.), dollar sign ($), underscore (_), or question mark (?).
break	is usually a space, tab, or NEW LINE character.

MASM normally converts all lowercase symbols to uppercase, so wsub is functionally the same symbol as WSUB.

By default, MASM recognizes only the first eight characters of any symbol as unique. For example, by default, it does not distinguish between ASSEMBLE and ASSEMBLER.

The first column of Figure 13-2 shows some symbols. Others examples are

$Z	BT12	CON
.DATA	DATA	EXIT.1
FATAL_ER	FILE1	MESSAGE
START		

If you need uniqueness to more than eight characters, MASM can provide this with a switch, described in the *Advanced Operating System/Virtual Storage (AOS/VS) Macroassembler (MASM) Reference Manual*, annotated in Chapter 16. For the examples in this book, eight characters are enough.

You can't use a period (.) as a single-character symbol. The system reserves this symbol to mean the value of the current location in a program. For example, WBR .–1 means "Wide Branch to the current location minus 1." Nor can you use a number as the first character of a symbol name — e.g., 5TEST. Finally, even though it is permitted, don't use a question mark (?) as the first character of any symbol. The operating system defines a set of symbols (including system calls) that uses the ? as the first character of each symbol. All the symbols in Figure 13-1 that begin with a ? are system-defined symbols.

If you follow these rules you may still receive an occasional assembler error indicating a bad symbol. This may happen because there are symbols, such as error codes that begin with ER (e.g., ERFDE) defined in the system parameter file. You can always clear up such errors by simply modifying the offending symbol, perhaps by placing a period or dollar sign at its beginning or end.

Argument Operators

From time to time, you may want the assembler to perform some operations on symbols. Here are some operators that you can use with symbols, integers, and the current location counter (.):

Type	Operator	Meaning
Bit	S or B	Set one or more bits. Use S for 32-bit values, B for 16-bit values. For example, `FOO: 1S0+1S7` defines value FOO with bits 0 and 7 set to 1 and other bits to 0.
Arithmetic Operators	+	Addition (e.g., 2+3).
	−	Subtraction (e.g., 5−2).
	*	Multiplication (e.g., 5*2).
	/	Division (e.g., 5/2).
Logical Operators	&	Logical AND.
	!	Logical OR.
	~	Logical NOT

Bit operators are useful when you want to check and compare bit values at runtime. For example, having set some bits i n a value, you can use runtime AND or OR instructions to check these new values and redirect control.

Arithmetic operators are useful with address offsets. If your program reserves a series of addresses whose first word is the address TABLE, offsets in TABLE would be TABLE+1, TABLE+2, etc. If NAME is the name of the beginning of a text string, NAME*2 points to the byte at the beginning of this string, a concept the next chapter explains in detail.

In terms of precedence, if you use more than one operator, MASM evaluates the bit operator first, then performs operations within parentheses left to right, then operations outside parentheses left to right. To have MASM compute the number of bytes between addresses TABLE and LBUFF, you would write `(TABLE-LBUFF)*2`.

Numbers

All numbers that you use in your programs are octal unless you specify otherwise. Indicate decimal values by placing a decimal point after the number. Thus 10 equals octal 10 (decimal 8), whereas 10. equals decimal 10 (octal 12).

Accumulators (Registers)

All ECLIPSE MV/Family computers have four 32-bit registers called accumulators (abbreviated AC). We call these AC0, AC1, AC2, and AC3; and in instructions you refer to them as 0, 1, 2, or 3. Accumulators are often used to store addresses and are always used for address and number manipulation and comparison. All four accumulators work much the same way, except that AC2 and AC3 provide address indexing; for example,

```
XWLDA    0,    -10, 2
```

means "load AC0 with the contents of the location found by calculating the value in (the contents of AC2 – 10_8."

Instruction Types

Each Data General storage word is 16 bits long. Most ECLIPSE MV/Family machine instructions assemble into one or two storage words. There are three types of instruction:

1. *Memory Reference (MRI) Instructions.* The MRIs concern a memory location or its content. They jump around and load and store addresses and values. There are four variations of each Load and Store instruction. The one you choose will depend on the width of the item you want to load (16 or 32 bits) and how far from the current location the item is. Some of the most useful MRI instructions, detailed in Table 13-1, are

   WBR
   XJMP, LJMP
   XJSR, LJSR
   XWLDA, LWLDA, XNLDA, XNLDA
   XWSTA, LWSTA, XNSTA, XNSTA
   XLEFB, LLEFB
   NLDAI, WLDAI

   Each MRI instruction requires that you specify a unique location in memory, an *addr*. You can do this in one of the following ways:

   - With absolute addressing — for example, `XJMP 25`, where `25` is the addr.
   - With program counter relative addressing — for example, `XJMP .+2` or `XJMP ERROR`, where `.+2` and `ERROR` are the addr locations, and where ERROR is a label in your program whose relative location the assembler will automatically compute.
   - With accumulator relative addressing — for example, `XJMP 1,3` or `XWLDA 0,10,2`, where 1 + contents-of-AC3 and where 10_8 + contents-of-AC2 are the addr locations.

   You can use symbolic addresses, values, and expressions in all three of these forms of addressing. The simplest form of MRI uses program counter relative addressing: you specify a label and the assembler computes the displacement from the current location. We show only this simplest form in program examples. But later you can use the other forms — especially accumulator relative addressing — as you become more experienced.

2. *Arithmetic-Logical Class (ALC) Instructions.* The ALC instructions can add and subtract values, shift bits, use the overflow (Carry), and redirect program execution. All ALC instructions require a source accumulator and a destination accumulator to receive the result of the arithmetic-logical operation. Some common ALC instructions are shown in Table 13-2.

3. *Input-Output (I/O) Instructions.* The I/O instructions govern the operation of all system devices. Generally, operating system calls will manage I/O devices, and you'll need these instructions only to write your own interrupt handlers.

Table 13-1. Common MRI Instructions

Instruction and Format	Explanation	Example
WBR addr	This instruction branches (jumps) to address addr. Accumulator relative addressing is not allowed with WBR. WBR assembles into one word and its addr must be within 127 words forward or 128 words backward. WBR is essential for system calls.	`WBR  ERROR`
XJMP addr LJMP addr	These instructions cause a jump to addr. For XJMP the addr can be up to 32 Kwords (32,767) locations away; it assembles into two words. For LJMP addr can be up to 256 Mwords (256 million words) away; it assembles into three words.	`XJMP  OUT` `LJSR  SUB`
XJSR addr LJSR addr	XJSR jumps and saves the return address; it is normally used to enter a subroutine. The subroutine can save the address via WSSVR 0, and later return via WRTN. The addr range and assembled size are the same as for XJMP and LJMP.	`XJSR  MY_SUB`
XWLDA ac addr LWLDA ac addr	These instructions load accumulator (ac) with two words, starting with the word at location addr. XWLDA works with an addr up to 32 Kwords away and assembles into two words. LWLDA works with an addr up to 256 Mwords away and assembles into three words. Second character W specifies a wide load.	`XWLDA  0, FNAME` `LWLDA  0, FAR  2`
XNLDA ac addr LNLDA ac addr	These load only one word at location addr but otherwise work the the same way as the wide-load instructions.	`XNLDA  1, FNAME` `LNLDA  1, FAR`
XWSTA ac addr LWSTA ac addr	These store the contents of ac in two words, starting with location addr. As with load, the XWSTA addr can be up to 32 Kwords away, assembling into two words; the LWSTA addr can be up to 256 Mwords away, assembling into three words. The second character W specifies a wide store.	`XWSTA  0, FNAME` `LWSTA  1, FAR`
XNSTA ac addr XNSTA ac addr	These store only the far right word (16 bits) of ac in addr (narrow store), but otherwise they work the same way as the wide store instructions.	`XNSTA  1, FNAME` `LNSTA  1, FAR`
XLEFB ac addr*2 LLEFB ac addr*2	These load ac with the first byte in addr. The load-effective byte instructions are very handy because byte addresses (byte pointers) are used extensively in system calls. After loading the byte address, you can store it via an XWSTA (or variation.) As with the other loads, the XLEFB addr can be up to 32 Kwords away, assembling into two words; the LLEFB addr can be up to 256 Mwords away, assembling into three words.	`XLEFB  0, BUFF*2`

NLDAI value ac
WLDAI value ac
	These instructions load value into ac (narrow and wide load immediate). The value can be an integer constant, symbolic expression of an integer constant, or ASCII characters. Use NLDAI for 16-bit values or two characters that assemble into two words. Use WLDAI for 32-bit values or four characters that assemble into two words.	`NLDAI  22.*3, 0`
		`NLDAI  'NO', 2`
		`WLDAI  NUM, 2`
		`WLDAI  'Test', 3`

Table 13-2. Common ALC Instructions

Instruction and Format	Explanation	Example
WADD acs acd	This instruction adds the 32-bit integer value in acs to the 32-bit integer value in acd. It puts the sum in acd.	`WADD 0, 2`
WSUB acs acd	This instruction subtracts the 32-bit integer value in acd from the 32-bit integer value in acs. It puts the result in acd.	`WSUB 2, 2`
WMOV acs acd	This copies the contents of acs to acd.	`WMOV 0, 2`
WSNE acs acd	This instruction compares acd to acs and skips the next word if the accumulators differ. If the two match, it doesn't skip. To test an ac for 0, make both acs and acd the same accumulator; the skip will occur if its value isn't 0.	`WSNE 1, 2` `WSNE 2, 2`
WSEQ acs acd	This instruction compares acd to acs, and skips the next word if the accumulators match. If they differ, they don't skip. To test for 0, make both acs and acd the same accumulator; the skip will occur if its value is 0.	`WSEQ 1, 2` `WSEQ 2, 2`

Indirect Addressing

In an MRI load or store instruction, you can use a commercial at sign (@). An @ before the addr argument in an MRI argument sets bit 16 in that instruction, the indirect addressing bit. For example,

```
000020 UC 101451  000060   XNLDA  0, 60
000022 UC 101451  100060   XNLDA  0, @60
```

The first instruction loads AC0 with the contents of memory location 60; the second loads AC0 with the word whose *address* is in location 60.

Pointers and "Range" Errors

Under some circumstances, you will want to use one symbol as a pointer to another symbol. Usually you will do this to overcome assembler "out of range" errors with instructions that have limited range, specifically WBR.

For example, you would get an out of range error with the following instructions:

```
        ?OPEN  FILE
        WBR  ERROR
        .               ; Code
LOOP:   .               ; here
        .               ; uses
        XJMP  LOOP      ; more
        .               ; than 127
        .               ; locations.
        .
ERROR:  .               ; Error routine.
        .
```

To fix the "range" error, use a pointer of the form

```
new-symbol:   XJMP   old-out-of-range-symbol
```

and branch to the pointer instead of the old symbol. To fix the example above, use the following code:

```
        ?OPEN  FILE
        WBR  ERR        ; Use the pointer.
        .               ; Code here
LOOP:   .               ; uses
        .               ; less
        .               ; than
        XJMP  LOOP      ; 127 locations.
ERR:    XJMP  EROR      ; Pointer.
        .
        .
        .
EROR:   .               ; Error routine.
        .
        .
```

Generally, you'll encounter this problem only with WBR.

Pseudo-ops

A pseudo-operative (pseudo-op) instruction directs the operation of the assembler. This instruction is called pseudo-operative because your program never executes it. Most of the pseudo-ops that you will need for assembly language programming are the following:

Pseudo-op	Function
.BLK	Reserves a block (series) of words.
.DWORD	Stores the value of the expression in two words. (MASM reserves two words by default.)
.END	Ends a module, and (optionally) names a starting address.
.ENT	Declares a symbol defined in this module to be available for use by another module. (Means enter.)
.EXTN or .EXTL	Declares a symbol external. (It's defined in some other module).
.LOC	Declares a location for the following data or code.
.NREL	Assembles the following code and data for execution in unshared memory. (Means normal relocatable.)
.TITLE or .TITL	Assign a title to a module.
.TXT	Stores an ASCII text string.
.WORD	Stores the value of the expression in one word.
.ZREL	Assemble the following code for execution in lower page zero. (Means zero relocatable.)

The .BLK pseudo-op is needed to establish space to receive filenames and for use as buffers.

The .DWORD pseudo-op stores a value in two words, whereas the .WORD pseudo-op stores a value in one. If you don't use either .WORD or .DWORD, MASM provides two words by default. You can change the default with the .ENABLE pseudo-op, described in the *Advanced Operating System/Virtual Storage (AOS/VS) Macroassembler (MASM) Reference Manual.*

The .ENT pseudo-op declares that an external symbol is defined in this module; .EXTN and .EXTL declare that a symbol is defined *outside* this module. If you ever plan a program that will use more than one module, you will want to define certain routines once, then have other modules use them. These pseudo-ops allow this.

The .NREL pseudo-op starts assembly storage at the beginning of unshared NREL memory. The start of NREL storage varies, as determined by Link. (.NREL can also specify shared memory.)

The .TXT pseudo-op assembles a text string into its equivalent ASCII codes (two per 16-bit word), and is terminated by a null byte. You can use any character not in the string as delimiters. Thus,

```
.TXT  "Abcde"
```

stores the ASCII codes for A and b in the first word, c and d in the second word, and e and <null> in the third and final word.

The .TITLE (or .TITL) pseudo-op assigns a title to a module. This title has no direct relationship to the module's filename. The title appears on the top line of each page of the module's assembly listing.

The .ZREL pseudo-op starts assembly storage from the first available ZREL location (location 50_8 by default). ZREL (also called lower page zero) storage extends from 50_8 to 377_8 and is directly addressable by all instructions.

.BLK

Reserves a block of storage.

Format

.BLK expression

Description

This pseudo-op reserves a block of memory. The expression is the number of words that you want reserved; the current location counter is incremented by the expression. The .BLK pseudo-op is often used for buffers, system parameter packets, or for any space into which you want to write something.

Example

The following example reserves a block of 60_{10} memory words (120_{10} bytes or characters). The first word in the series has the symbolic name TABLE:

```
TABLE:          .BLK 60.        ; Block for TABLE.
```

The next example reserves a block of ?IBLT memory words; the first word has the name PAKET. This is the beginning of a system parameter packet:

```
PAKET:          .BLK    ?IBLT ; Block for packet.
```

.DWORD

Stores the value of the expression in two words.

Format

.DWORD expression

Description

The .DWORD pseudo-op directs MASM to store the value of expression in two words. The expression is often a number or the name of a location.

In this chapter we choose to explicitly define storage with the .DWORD and .WORD pseudo-ops. But you don't have to. By default, MASM gives each expression two words. You can also define storage globally with the .ENABLE pseudo-op, defined in the *Advanced Operating System/Virtual Storage (AOS/VS) Macroassembler (MASM) Reference Manual.*

Generally, the only packet locations that require two words are byte and word pointers. Most system call parameter packet symbols require one (and only one) word that you can specify with the .WORD pseudo-op, described later.

Example

This example has all executable code preceding the system parameter packet. It uses .DWORD or .WORD whenever necessary to assign storage.

```
          .TITLE  MPROG
          .NREL

START:    ?OPEN  CON
           .
           .                    ; Program code.
           .
          ; Parameter Packet

CON:      .BLK    ?IBLT         ; Length of packet.
          .LOC   CON+?ISTI      ; Loc for stats.
          .
          .
          .
          .LOC   CON+?IBAD      ; Loc for byte pointer.
          .DWORD  BUFF*2        ; 2 words for this.
          .LOC   CON+?IRCL      ; Location for record length.
          .WORD 120
          .
          .
          .
          .LOC   CON+?IBLT      ; End of packet.
BUFF:     .BLK 60.              ; Buffer.
```

See also the example for the .LOC pseudo-op, described later.

.END

Indicates the end of a module.

Format

.END *[expression]*

Description

This pseudo-op terminates each assembly language module. At least one module in each program must supply an *expression* argument to the .END pseudo-op, indicating the address that will receive control when the program is executed.

Example

In this example, the .END pseudo-op terminates the module and defines the address that will receive control when you run the program.

```
START:          ?OPEN  CON        ; Open console.
                .
                .
                .
                .END  START
```

.ENT

Defines a module entry.

Format

.ENT symbol *[,symbol] ...*

Description

This pseudo-op declares user symbols defined within the module to be available for reference by other, separately assembled modules. Each **symbol** must be unique from other symbols within the module (this is always a requirement). It must also be unique among entries defined with **.ENT** in other modules that you will link together to form a single program.

Separately assembled modules that will use the .ENT pseudo-op must declare the **symbol** external, using the .EXTN or .EXTL pseudo-op.

Example

In the following example, PROGA is the main program. It can call two external routines in module TRIG: SINE and COS.

The CLI commands that assemble and Link these program modules, forming program PROGA.PR, might be

```
) xeq masm proga ↵

) xeq masm trig ↵

) xeq link proga trig ↵
```

```
        .TITL   PROGA   ; PROGA is main module.
        .EXTN   SINE    ; SINE is external normal.
        .EXTL   COS     ; COS is external long.
        .NREL
START:  .               ; PROGA gathers data for computation
        .               ; until it needs
        .               ; routine SINE or COS (cosine).
        XCALL   SINE    ; XCALL to .EXTN symbol in other module.
        .               ; Return here and continue.
        .
        .
        LCALL   COS     ; LCALL (long) to .EXTL symbol in other module.
        .               ; Return here and continue.
        .
        .
        .END    START


        .TITLE  TRIG    ; TRIG does trig.
        .ENT SINE, COS  ; Identify entries.
        .NREL
```

.ENT (continued)

```
SINE:   WSAVR   0       ; Save return address with WSAVR.
        .               ; SINE processing routine is here.
        .
        .
        WRTN            ; Done with SINE code, WRTN to calling module.
        .
        .               ; For sake of example, assume 32 Kbytes or so
        .               ; of code here (to justify .EXTL in caller).

COS:    WSAVR   0       ; COS routine also saves return.
        .               ; Do cosine processing.
        .
        .
        WRTN            ; Return to calling program module.
        .END
```

.EXTL

Declares an external long reference.

Format

.EXTL symbol *[,symbol] ...*

Description

The .EXTL pseudo-op declares that one or more symbols used by this module are defined (with the .ENT pseudo-op) in some other module.

Use .EXTL for symbol(s) that will be defined with 256 Mword locations in the finished program. Use .EXTN, described next, for symbol(s) that will be defined within 32 Kword locations in the finished program.

Each "symbol" that you name in an .EXTL pseudo-op must be defined with an .ENT (entered) pseudo-op in another module that you will link along with this module. If an .EXTL symbol is undefined, the Link utility reports an error when it builds the program file.

Example

See the immediately preceding .ENT example.

.EXTN

Declares an external normal reference.

Format

.EXTN symbol *[,symbol] ...*

Description

The .EXTN pseudo-op declares that one or more symbols used by this module are defined (with the .ENT pseudo-op) in some other module.

Use .EXTN for symbol(s) that will be defined within 32 Kword locations in the finished program. Use .EXTL, described earlier, for symbol(s) that will be defined with 256 Mword locations in the finished program.

Each "symbol" that you name in an .EXTN pseudo-op must be defined with an .ENT pseudo-op in another module that you will link along with this module. If an .EXTN symbol is undefined, the Link utility reports an error when it builds the program file.

Example

See the .ENT example, shown earlier.

.LOC

Sets the location counter.

Format

.LOC expression

Description

The .LOC pseudo-op sets the location counter value to **expression**, where **expression** is any kind of numeric statement — for example, A+B, where A and B have assigned values.

Generally, for .NREL (normal relocatable) code, you should use only symbols, not constants, within **expression**. Using symbols makes your program easier to understand, and much easier to update if you ever need to change portions of it. Also, for all system constructs (like parameter packets), the system defines symbols that contain correct constants, as you will see in the next chapter.

Example

The following example shows how the .LOC pseudo-op changes the value in the location counter. The symbols each have parameter values (the whole example is a system parameter packet). You can see how, used with the .LOC pseudo-op, symbols move the location counter up through 000176, back to 000165, then up to 000200.

Note that the location counter moves from 000162 to 000200 for the packet, although the packet specifically defines only 11_8 words. This shows how the system-defined symbols allot needed space — reserving locations that the system needs, even though the user doesn't need or care about them.

The following is an assembled listing.

```
10
11                              ; Console I/O packet.
12
13 000162 UC 000000 000030      CON:    .BLK    ?IBLT     ; Packet length.
14                                      .LOC    CON+?ISTI ; Offset for statistics.
15 000163 UC 040032                     .WORD   ?ICRF+?RTDS+?OFIO
                                                          ; Change fmt+D-S rec+I&O.
16                                      .LOC    CON+?IBAD ; Offset for Buffer ADdr.
18                                      .DWORD  BUFF*2    ; 2 wd bptr to I/O buffer.
                                        .LOC    CON+?IRCL ; Offset for max ReC Len.
19 000171 UC 000170                     .WORD   120.      ; Max rec length of 120 chars.
20                                      .LOC    CON+?IFNP ; Offset for Filename Ptr.
22                                      .DWORD  CNAME*2   ; 2 wd bptr to con filename.
                                        .LOC    CON+?IMRS ; Offset for MemoRy Size.
23 000165 UC 177777                     .WORD   -1        ; Memory block size, default.
24                                      .LOC    CON+?IDEL ; Offset for DELimter addr.
26                                      .DWORD  -1        ; D-S del table addr, defautl.
                                        .LOC    CON+?IBLT ; End of packet.
```

.NREL

Sets the location counter to unshared code.

Format

.NREL

Description

The .NREL pseudo-op sets the location counter to the first word of unshared, relocatable memory. This pseudo-op is normally required in each module. The .NREL pseudo-op has other uses, described in the MASM reference manual.

Example

```
        .TITL   SUB_PROG
        .EXTN   MAIN, ERR
        .ENT    SUBR, SUBR1

        .NREL   ; Normal relocatable code.
INIT:   .
        .
        .
        .END
```

See also the .ENT example, shown earlier.

.TITLE

Provides a name for an object module.

Format

.TITLE symbol

Description

.TITLE gives an object module a title (symbol), which is printed at the top of every listing page. The title need not be different from other symbols in the module. The title has no inherent relationship to the module's filename, but most people use the same names for clarity. If you omit symbol, MASM assigns the title .MAIN.

Example

This example shows how .TITLE assigns the title EXAMPLE to a module and will, subsequently, print EXAMPLE at the top of each listing page.

```
        .TITLE  EXAMPLE
        .NREL
BEGIN:  .
        .
        .
        .END    BEGIN
```

.TXT

Creates a text string.

Format

.TXT u string u

Description

This pseudo-op creates an ASCII string. You must delimit the text with a character that is unique, u — that is, not found in the string. Most often, u is a quotation mark or a backslash. You can put nonprinting characters in the string by enclosing their ASCII values in angle brackets.

If you insert a NEW LINE character (ASCII 12_8) in the string, the system may treat it like a record delimiter, and not write the rest of the line. To make sure the system writes the entire line, simply add 200_8 to the NEW LINE ASCII value; for example,

```
.TXT "This<212>
 is a multiline<212>
 text string.<12>"
```

Examples

The following example is an ASCII string consisting of three words. The first word is the letters Ab, the second, the letters cd, and the last word, an e in the left byte and a terminating null in the right byte. The text delimiters are quotation marks.

```
.TXT  "Abcde"
```

The next example shows the ASCII string consisting of the letters Ab and cde, separated by a horizontal tab (ASCII 11_8), and delimited by quotations marks.

```
.TXT  "Ab<011>cde"
```

.WORD

Stores the value of the expression in one word.

Format

.WORD expression

Description

The .WORD pseudo-op directs MASM to store the value of expression in one word.

In this chapter we choose to explicitly define storage with the .DWORD and .WORD pseudo-ops. But you don't have to. You can define storage globally with the .ENABLE pseudo-op, defined in the *Advanced Operating System/Virtual Storage (AOS/VS) Macroassembler (MASM) Reference Manual.* By default, MASM gives each expression two words.

Most system call parameter packet symbols require one (and only one) word that you specify with .WORD. Generally, the only packet locations that require two words are byte and word pointers.

Example

See the example for the .DWORD pseudo-op, shown earlier.

.ZREL

Sets the location counter to lower page zero.

Format

.ZREL

Description

This pseudo-op sets the location counter to the first available word in lower page zero. This is usually location 50_8.

Example

The following example shows how MASM maintains both a ZREL and NREL location counter. It also shows that you can change from ZREL to NREL, and vice-versa, at will.

```
01                                .TITLE Z
02                                .ZREL   ; Assemble
03                                        ; for ZREL.
04
05 000000  ZR  000000 000003     3       ; Value of 3.
06 000002  ZR  000000 000004     4       ; Value of 4.
07
08                                .NREL   ; Assemble
09                                        ; for NREL.
10 000000  UC  000000 000003     3       ; Value of 3.
11 000002  UC  000002 000004     4       ; Value of 4.
12
13                                .ZREL
14 000004  UC  000000 000006     6       ; Value of 6.
15 000006  UC  000000 000007     7       ; Value of 7.
```

What Next?

This chapter has given you some of the fundamentals about assembly language, on the Macroassembler, assembly language statements, the assembler listings, symbols, instructions, and pseudo-ops.

After reading this chapter, you have the background needed for the next chapter: Writing AOS/VS Assembly Language Programs.

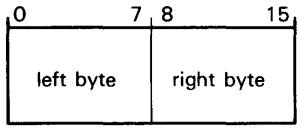
End of Chapter

Chapter 14
Writing AOS/VS Assembly Language Programs

This chapter describes words and bytes, and then explains the format and shape of system calls. After that, you will write and assemble an assembly language program, correct the assembly errors, link it, execute it, and finally, debug it.

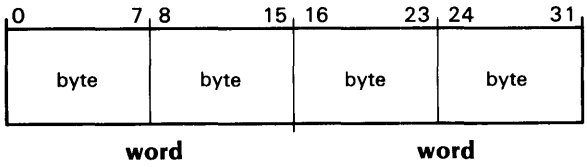
Words and Bytes

An ECLIPSE computer word is 16 bits long, and its bit positions are numbered left to right, from 0 through 15. A byte is 8 bits long. A text string consists of a sequence of bytes, packed left to right in a series of one or more words.



ID-03316

The accumulators, some instructions, and most user symbols use two words (32 bits). You can think of these as either 32-bit entities, two-word entities, or four-byte entities, whichever is most meaningful.



ID-03314

Byte Pointers

System calls often use byte pointers, which are pointers to byte addresses. You can form a byte pointer by multiplying the starting location of the byte by 2. You can load a byte pointer with the XLEFB instruction (mentioned in the last chapter); and you can store one with the appropriate store instruction. For example,

```
        XLEFB  0, TEXT*2        ; Byte pointer is
                                ; TEXT*2.
        XWSTA  0, STORE         ; Store byte
        .                       ; pointer.
        .
        .
STORE:  .DWORD  0               ; 2 words for wide store.
TEXT:   .TXT   "This is a text string."
```

System Call Format

Each system call is an assembly language macro, supplied by Data General. You call it using the following format:

```
?call-name
exception (error) return
normal return
```

Since the ?call-name is a macro, the macroassembler will expand it during assembly. Depending on the call, it will expand into 2 to 6 locations. Therefore, program listings show 2 to 6 locations used for each system call.

The exception (error) return will receive control on either an error condition, or an exceptional condition such as an end-of-file. The exception error return must be a *one-word* instruction that tells the system what to do if an error occurs: for example, `WBR ERROR` where `ERROR` starts an error-handling routine. Because the exception instruction must be one word long, you can't use long-range MRIs such as XJMP for it — because these assemble into two or more words. You must always reserve an exception return word, even if you expect no exception conditions.

Whenever control goes to the exception return, the system places a numeric error code in AC0. You can get an English language explanation of any octal code from the CLI by typing

```
) message code-number ↵
```

Some common error codes are described later in this chapter.

If the call executes normally, control goes to the normal return and the program continues.

System calls that involve I/O take the address of a parameter packet as an argument; for example,

```
?call-name   packet-address
exception (error) return
normal return
```

When AOS/VS executes a call that involves I/O, it places the parameter packet address in AC2 before executing the call. Thus, for I/O calls, AC2 is generally overwritten. Upon either an exception return or a normal return, AC3 contains the current contents of locations 20_8 and 21_8 (the wide frame pointer, WFP); thus AC3, too, is overwritten.

Generally, do not use memory locations below 50_8: the system uses these locations for call processing.

Operating System Calls

We describe here only the most important features of operating system calls, in the order the sample program uses them. Later, you may want to use other features. For many applications, however, these versions will suffice:

?OPEN	Opens a file or device.
?READ	Reads a record: for example, a line of text.
?WRITE	Writes a record: for example, a line of text.
?GTMES	Reads from the CLI command line.
?RETURN	Stops the program and returns to the CLI.

The system call ?OPEN opens a file, or a device like the terminal, for access. The ?OPEN call can also create a file.

The ?READ call reads information from the file, and the ?WRITE call writes it to the file.

The ?GTMES call lets the program read arguments from the CLI command line that executed the program; for example, the program reads FIX from the command line XEQ MPROG FIX.

The ?RETURN call closes all open files, terminates the program, and returns to the CLI. ?RETURN can also help identify errors by sending explanatory error messages to the terminal.

The sample program we will discuss uses all these calls. It uses ?GTMES to get a disk filename, then uses ?OPEN to open the disk file, creating it if necessary. Then it uses ?READ to read lines entered from the keyboard and write (with ?WRITE) the lines back to the screen and to the disk file. It uses ?RETURN both to handle errors and to return normally to the CLI.

?OPEN

Opens (optionally creates and opens) a file.

Format

```
?OPEN   packet-address
exception (error) return
normal return
```

Input and Return

Input: Aside from packet-address, none.

Return: On any return, AC2 contains the packet address. On an exception return, AC0 contains an error code, described later in this chapter.

Description

You must open a file before you read or write records in the file. For each of the system calls in the I/O sequence —?OPEN, ?READ, ?WRITE — you must supply a parameter packet. You can use the same I/O packet to open, read, and write to a file. Some parameters apply only to ?OPEN, others to ?READ and ?WRITE, but it's easiest to set all the parameters at the beginning and to change only a few parameters, such as byte pointers, at runtime.

An I/O parameter packet can specify such instructions as

- Change a file's record format, if needed.
- Create a file if it doesn't exist. If it does exist, the ?OPEN call can delete and recreate it, or open it for appending, so all writes will go to the end of the file, making it longer.
- Allow input, output, or both.
- Create data-sensitive records, which means that one or more special characters will be record delimiters.
- Use a given buffer for I/O, if any.
- The filename (pathname) to open.
- The maximum record length for each read or write.

All parameter packets should be built parametrically. This means that instead of using numeric constants in packets, you should use system-defined symbols. For example, ?IBLT is the symbol that specifies the length for every I/O packet. Other system symbols define offsets that, used with the .LOC pseudo-op, will automatically set up the correct values.

The I/O packet used for ?OPEN, ?READ, and ?WRITE is ?IBLT words long. Figure 14-1 shows both the contents of each packet offset and a sample packet for the console. In Figure 14-1, pkt-addr means packet-address. Each symbolic location in the packet requires one word, unless shown otherwise.

The symbols that set bits must follow the .LOC that sets the location. Notice in Figure 14-1 that `.LOC FILE+?IBLT` is last.

```
Name of Offset          Contents


pkt-addr:               .BLK  ?IBLT means reserve ?IBLT words for packet.

pkt-addr+?ISTI          ?ICRF means adjust the record format.
                         +?IFCE means create the file, if needed.
                         +?APND means open for appending.
                         +?OFIO means open for input and output.
                         +?RTDS means data-sensitive records.

pkt-addr+?IBAD          Two-word byte pointer to I/O buffer --  for
                        example, .DWORD  BUF*2.

pkt-addr+?IRLR          0. You can omit this if you want.
                        In this location, the system returns the number
                        of bytes read or written after ?READ or ?WRITE.

pkt-addr+?IRCL          The maximum record length -- for example, 120,
                        which means a maximum length of 120 characters.

pkt-addr+?IFNP          Two-word (.DWORD) byte pointer to filename
                        (pathname) -- for example, .DWORD  NAME*2.

pkt-addr+?IMRS          -1.  Aside from setting this to -1, you
                        can ignore it.

pkt-addr+?IDEL          Two-word (.DWORD) pointer to delimiter table
                        address.  Aside from setting this pointer to
                        -1, you can ignore it.

A real I/O packet, and the filename and I/O buffer:


FILE:   .BLK    ?IBLT               ; Reserve ?IBLT words.
        .LOC    FILE+?ISTI          ; Location "I/O statistics."
        .WORD   ?ICRF+?APND?OFCE+?OFIO+?RTDS
                                    ; Change record format+append+
                                    ; create+I&O+d-s recs.
        .LOC    FILE+?IBAD          ; Location "I/O buffer address".
        .DWORD  BUF*2              ; Byte pointer to buffer.
        .LOC    FILE+?IRLR          ; Location "I/O record length returned".
        .WORD   0                   ; Sys returns number of bytes transferred.
        .LOC    FILE+?IFNP          ; Location "I/O filename pointer."
        .DWORD  NAME*2              ; Byte pointer to filename (pathname).
        .LOC    FILE+?IMRS          ; Location "I/O memory size."
        .WORD   -1                  ; Default with  -1.
        .LOC    FILE+?IDEL          ; Location "I/O" delimiter".       "
        .DWORD  -1                  ; Use -1 for standard delimiters.
        .LOC    FILE+?IBLT          ; End of I/O packet.

BUF:    .BLK    60.                 ; 60. word I/O buffer.

NAME:   .TXT    "MYFILE"           ; Filename.
```

Figure 14-1. Assembly Language I/O Packet

?READ and ?WRITE

Reads and Writes a record.

Format

```
?READ   packet-address
exception (error) return
normal return
```

or

```
?WRITE   packet-address
exception (error) return
normal return
```

Input and Return

Input: Aside from packet-address, none.

Return: On any return, AC2 contains the packet address. On an exception return, AC0 contains an error code, described later in this chapter.

Description

You issue ?READ to read a record, and ?WRITE to write a record. When you first open a file, the system sets its file pointer to the beginning of the file. Each ?READ or ?WRITE transfers one record beginning at the current position of the file pointer. Then the system moves the file pointer to the beginning of the next record.

For details about the ?READ and ?WRITE packet, see the description given under the ?OPEN call.

?GTMES

Gets a message from the CLI.

Format

?GTMES packet-address
exception (error) return
normal return

Input and Return

Input: Aside from packet-address, none.

Return: ?GTMES can get one or more parts, and/or all, of the command line, depending on the packet. In most cases, AC0 and AC1 are overwritten with information. If an error occurred, AC0 contains an error code (described later in this chapter). On any return, AC2 contains the packet address.

Description

Each time you create a new process (as when you use the CLI XEQ command to execute a system utility or one of your own programs), the CLI sends a message to the new process. The ?GTMES system call lets you read this message.

The message contains a slightly edited version of the CLI command you typed to start the program: the XEQ command disappears from the edited CLI command line, and a comma replaces each instance of single or multiple spaces that separate command-line arguments. Thus if the CLI command you entered was

```
x myfile/qq=444 fix
```

the edited version of the message would be

```
myfile/qq=444,fix
```

MYFILE is argument number 0, /QQ=444 is the switch for argument 0, and FIX is argument 1.

The sample program later in this chapter uses the ?GTMES call to read the filename argument given in the XEQ command. The ?GTMES parameter packet in the program looks like this:

```
pkt-addr:  .BLK  ?GTLN            ; Reserve ?GTLN words.
           .LOC  pkt-addr+?GREQ   ; Location "Get Request".
           .WORD ?GARG            ; Contents: means "Get argument".
           .LOC  pkt-addr+?GNUM   ; Location "Get Number".
           .WORD  1               ; Contents (means "Arg number 1").
           .LOC  pkt-addr+?GSW    ; Location "Get Switches".
           .DWORD  0              ; Don't care about switches.
           .LOC  pkt-addr+?GRES   ; Location for byte pointer.
           .DWORD byte pointer    ; Two-word byte pointer to area
                                  ; that will receive ?GTMES info.
           .LOC  pkt-addr+?GTLN   ; End of packet.
```

?RETURN

Terminates this process.

Format

?RETURN
exception (error) return

Input and Return

Input: For a good (correct) return to the CLI, set AC2 to 0. For an exception return to the CLI, load AC2 with ?RFEC+?RFCF+?RFER.

Return: Generally, none, because the program that issues ?RETURN is terminated.

Description

This call terminates the calling process, and returns control to the CLI. If you want the program to return normally, simply set AC2 to 0 before issuing this call.

If the program is returning (with ?RETURN) because of an error, load AC2 with ?RFEC+?RFCF+?RFER before issuing the ?RETURN call. The CLI will then interpret the error code passed in AC0 and display the appropriate error message on the terminal.

There is no normal return because the call terminates the calling program.

Common Errors

The following errors are the most common ones that can occur on the system calls we have described. If one of them occurs, the system places the appropriate error number in AC0, then goes to the call's exception return.

Error Mnemonic	Meaning
EREOF	End of file.
ERFDE	Filename (pathname) does not exist.
ERFNO	File not open.
ERIFC	Illegal filename (pathname).
ERMPR	Error in the system call parameters. This often means a bad packet address.
ERVBP	This usually means that you made a mistake with the byte pointer code.

If you know that a call may encounter an error condition, such as a ?READ call encountering an end-of-file, you can have the call's error return go to an error processing routine. The routine would compare AC0 to the value of the error you wanted to trap, and then the routine could act on the result. For example,

```
        ?READ   FILE      ; Read from file.
        WBR     EOF?      ; Error, check it.
        .                 ; Normal return.
        .
        .
EOF?:   WLDAI   EREOF, 1  ; Put code for EOF
                          ; in AC1.
                          ; AC0 has current error.
        WSEQ    0,1       ; Skip if EOF.
        WBR     BYE       ; Other error, exit.
        WBR     CONT      ; Take other action if EOF.
        .
        .
        .
```

We recommend that you use symbol mnemonics, rather than constants, wherever possible.

Program Example

The following sample program, called the WRITE program, is a short assembly language program that uses two files: the terminal and a disk file that it creates.

The program first gets the filename from the XEQ command; then it opens the terminal and the specified file. (The terminal filename is @CONSOLE.)

Next, the program displays a prompt on the screen, reads a line from the keyboard, echoes the line on the screen, and writes the line to the specified file. Then it loops back to display the prompt again. When you type the characters NO ↵ after the prompt, the program terminates and returns to the CLI.

If an error occurs, the program reports the condition by placing the proper flags in AC2 and issuing a ?RETURN call.

Figure 14-2 is a flow chart of the WRITE program. Figure 14-3 is the assembled listing of the program, with errors. We have omitted the cross-reference listing for brevity.

Even if you decide not to try the program, you should examine the flow chart and program listing before continuing to the next section.

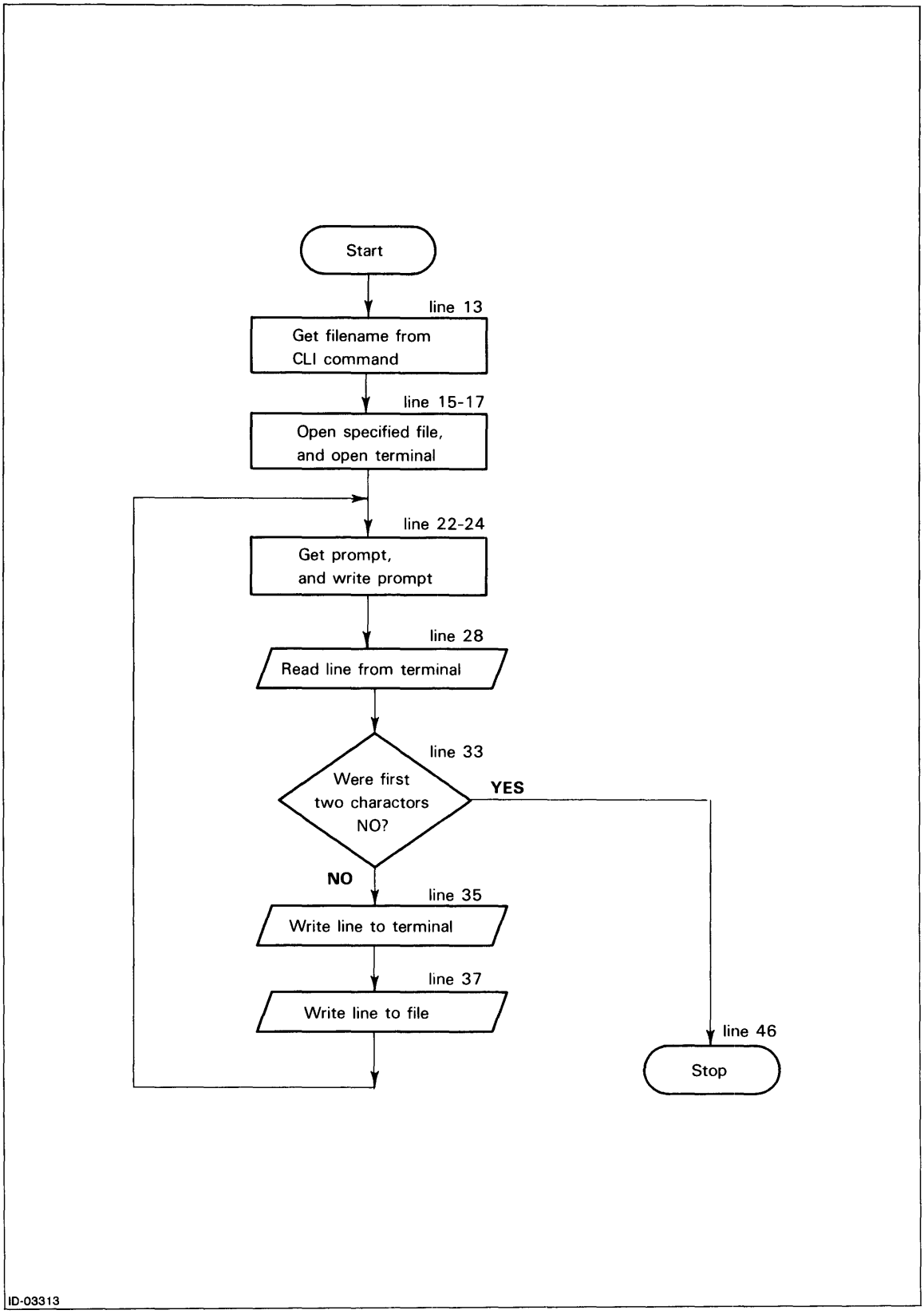


Figure 14-2. WRITE Program Flow Chart (Assembly Language)

```
SOURCE: WRITE            MASM 06.00.00.00       10-OCT-85 14:34:57  PAGE    1

01                              ;  This program gets a filename from the CLI XEQ
02                              ; command, opens the terminal and the specified file,
03                              ; writes a prompt to and reads a line from the
04                              ; terminal, and writes the line to terminal and file.
05                              ; To stop it, type NO and press NEW LINE.
06                              ; To run: XEQ WRITE <output-pathname>

07                                      .TITLE  WRITE
08                                      .ENT    WRITE     ; For debugging.
09                                      .NREL             ; Unshared partition.
10
11                              ; Get filename; open terminal and specified file.
12
13                              WRITE:  ?GTMES  CLIMS     ; Get the filename.
14 000006 UC 115270                     WBR     ERROR     ;   Error, process it.
15                                      ?OPEN   FILE      ; Open file, create if needed.
16 000015 UC 114370                     WBR     ERROR     ;   Error, process it.
17                                      ?OPEN   CON       ; Open console.
18 000024 UC 111470                     WBR     ERROR     ;   Error, process it.
19
20                              ; Write prompt, read line, check for NO, write files.
21
22 000025 UC 122071 000146      LOOP:   XLEFB   0, PROMPT*2  ; Get bptr to prompt.
23 000027 UC 121431 000137              XWSTA   0, CON+?IBAD ; Put in console pkt.
24                                      ?WRITE  CON          ; Write prompt to console.
25 000037 UC 110170                     WBR     ERROR        ;   Error, process it.
26 000040 UC 122071 000416              XLEFB   0, BUFF*2    ; Get bptr to I/O buffer.
27 000042 UC 121431 000124              XWSTA   0, CON+?IBAD ; Put in console pkt.
28                                      ?READ   CON          ; Read a line from console.
29 000052 UC 104670                     WBR     ERROR        ;   Error, process it.
30 000053 UC 143051 047117              NLDAI   'NO',  0     ; Put NO term. in AC0.
31 000055 UC 125411 000172              XWLDA   1, BUFF      ; Get first WORD (2
32                                                           ;  bytes) of buffer.
33 000057 UC 120611                     WSNE    1, 0         ; Skip if not equal NO.
34 000060 UC 104470                     WBR     GOODBYE      ;   User typed NO, exit.
35                                      ?WRITE  CON          ; Not NO. Echo line, go on.
36 000067 UC 101170                     WBR     ERROR        ;   Error, process it.
37                                      ?WRITE  FILE         ; Write line to file.
38 000076 UC 100270                     WBR     ERROR        ;   Error, process it.
39 000077 UC 164670                     WBR     LOOP         ; Do it all again.
40                              ;  Process error and/or return here.
41
42 000100 UC 153211             ERROR:  WLDAI   ?RFEC+?RFCF+?RFER, 2 ; Get err flgs.
43           00000150000
 E 000103 UC 155570                     WBR     BYE                  ; Skip subtraction.
45
46 000104 UC 150531             GOODBYE: WSUB   2, 2                 ; Set for good return.
47                              BY:     ?RETURN                      ; To CLI.
48 000110 UC 175070                     WBR     ERROR                ; ?RETURN error.
49
50                              ;  Prompt, terminated with NEW LINE.
51
52 000111 UC 040547 060551      PROMPT: .TXT  "Again?<12>"   ; Prompt.
53           067077 005000
```

Figure 14-3. Assembly Language WRITE Program with Errors (continues)

```
54
55                              ; Parameter packets and buffers.
56
57                                      .ENABLE  WORD ; Most pkt entries = 1 word.
58
59                              ; ?GTMES packet to get the message.




SOURCE: WRITE             MASM 06.00.00.00       10-OCT-85 14:34:57  PAGE    2

01 000115 UC  00000000006       CLIMS:  .BLK    ?GTLN           ; ?GTMES packet length.
02            00000000115  UC           .LOC    CLIMS+?GREQ     ; Offset to request type.
03 000115 UC 000003                     .WORD   ?GARG           ; We want to get an argument.
04            00000000116  UC           .LOC    CLIMS+?GNUM     ; Offset to arg NUM.
05 000116 UC 000001                     .WORD   1               ; Arg number 1 is filename.
06            00000000121  UC           .LOC    CLIMS+?GRES     ; Offset to GTMES buffer.
07 000121 UC  00000000246  UC           .DWORD  NAME*2          ; 2 wd bptr to NAME buffer.
08            00000000123  UC           .LOC    CLIMS+?GTLN     ; End of ?GTMES packet.
09
10 000123 UC  00000000040       NAME:   .BLK    32.             ; Reserve 64 chars for fname.
11
12                              ; Console I/O packet.
13
14 000163 UC  00000000030       CON:    .BLK    ?IBLT           ; Packet length.
15            00000000164  UC           .LOC    CON+?ISTI       ; Offset for statistics.
16 000164 UC 040032                     .WORD ?ICRF+?RTDS+?OFIO ; Change fmt+D-S rec+I&O.
17            00000000167  UC           .LOC    CON+?IBAD       ; Offset for buffer addr.
18 000167 UC  00000000520  UC           .DWORD  BUFF*2          ; 2 wd bptr to I/O buffer.
19            00000000172  UC           .LOC    CON+?IRCL       ; Offset for max rec length.
20 000172 UC 000170                     .WORD   120.            ; Max rec length of 120 chars.
21            00000000177  UC           .LOC    CON+?IFNP       ; Offset for file name ptr.
22 000177 UC  00000000426  UC           .DWORD  CNAME*2         ; 2 wd bptr to con filename.
23            00000000166  UC           .LOC    CON+?IMRS       ; Offset for memory size.
24 000166 UC 177777                     .WORD   -1              ; Memory block size, default.
25            00000000201  UC           .LOC    CON+?IDEL       ; Offset for delimter addr.
26 000201 UC  37777777777               .DWORD  -1              ; D-S del table addr; default.
27            00000000213  UC           .LOC    CON+?IBLT       ; End of console I/O packet.
28
29 000213 UC 040103 047516      CNAME:  .TXT "@CONSOLE"         ; Use generic name.
30           051517 046105
31           000000
32
33                              ; Open and I/O packet for FILE.
34
35 000220 UC  00000000030       FILE:   .BLK    ?IBLT           ; Packet length.
36            00000000221  UC           .LOC    FILE+?ISTI      ; Offset for statistics.
37 000221 UC 040272                     .WORD   ?ICRF+?APND+?OFCE+?RTDS+?OFIO
                                                                ; Change fmt+
38                                                              ; append+create+D-S recs+I&O.
39            00000000224  UC           .LOC    FILE+?IBAD      ; Offset for buffer addr.
40 000224 UC  00000000520  UC           .DWORD  BUFF*2          ; 2 wd bpt to I/O buffer.
41                                                              ; (Same as CON's).
```

Figure 14-3. Assembly Language WRITE Program with Errors (continued)

```
42            00000000227  UC          .LOC    FILE+?IRCL      ; Offset for max rec length.
43 000227 UC  000170                    .WORD   120.            ; Max record length of 120 chars.
44            00000000230  UC          .LOC    FILE+?IRLR      ; Offset for ret. len record.
45 000230 UC  000000                    .WORD   0               ; Sys returns # of chars transferred.
46            00000000234  UC          .LOC    FILE+?IFNP      ; Offset for file name ptr.
47 000234 UC  00000000246  UC          .DWORD  NAME*2          ; 2 wrd bptr to NAME,
48                                                            ; obtained by ?GTMES.
49            00000000223  UC          .LOC    FILE+?IMRS      ; Offset for memory size.
50 000223 UC  177777                    .WORD   -1              ; Memory block size, default.
51            00000000236  UC          .LOC    FILE+?IDEL      ; Offset to delimter addr.
52 000236 UC  37777777777               .DWORD  -1              ; D-S del table addr, default.
53            00000000250  UC          .LOC    FILE+?IBLT      ; End of file packet.
54                                 ; I/O Buffer. At end to avoid WBR addr errors.
55
56 000250 UC  00000000074       BUFF:   .BLK 60.                ; Buffer of 60. words, 120. chars.
57
 E                                      .END    .WRITE          ; Start at the beginning.




ERRORS: WRITE             MASM 06.00.00.00       10-OCT-85 14:34:57  PAGE    4

pass 2 errors:
        WRITE1.SR    1/43: listing   1/44: symbol is undefined
        WRITE1.SR    1/43: listing   1/44: illegal relocation
        WRITE1.SR    1/114: listing   2/58: symbol is undefined

3 ASSEMBLY ERRORS
```

Figure 14-3. Assembly Language WRITE Program with Errors (concluded)

Writing and Assembling the WRITE Program

If you want to try this program, we suggest that you first create a directory for it. This will prevent conflicts with other programs that have the same name and will encourage you to place all your assembly language programs in the same place. For example, type

```
) dir/i ↓
) cre/dir asm ↓
) dir asm ↓
```

Now, execute the SED or SPEED text editor and create a file named WRITE.SR. The .SR suffix identifies the program as the source file.

```
) xeq sed write.sr ↓
```

or

```
) xeq speed/d write.sr ↓
```

Type in the program according to Figure 14-3. This is an assembled listing so *don't* type in any of the material in columns 1–32, the shaded part of the figure. The code on the right is what you enter.

You can type in the program in either uppercase or lowercase. One approach is to put comments and .TXT strings in lowercase, everything else in uppercase, as in Figure 14-3.

If you want your listing numbers to match those in Figure 14-3, put line 44 of page 1 on the line *directly* after line 1-42; for example,

```
ERROR:    WDLAI ... ↓
          WBR ... ↓
```

The assembler will insert the third word of the WLDAI instructions between these lines.

After typing in the WRITE program, leave the text editor and assemble the program with the command,

```
) xeq masm/l=@lpt write ↓
```

During the assembly, MASM reports several errors on the screen. From the line printer, you will get a listing very similar, if not identical, to the one shown in Figure 14-3. You will also get an assembler cross-reference, which we have omitted.

Analyzing the WRITE Program

In this analysis of the WRITE program we examine the code line by line, starting on page one.

On line 8, the program .ENT pseudo-op defines the starting address not because other modules will use it, but because we want to identify the symbol WRITE to the debugger. Later this will help us debug the program.

On line 9, the .NREL pseudo-op specifies normal relocatable, unshared code.

On lines 13 through 18, the code gets the output file pathname from the CLI command line, opens this file, and opens the terminal (generic name @CONSOLE). The ?GTMES call will get the output file's pathname from the XEQ command we use to run the program. On line 16, is the exception return: the WBR ERROR. We process all error conditions in this program by branching to ERROR, described later.

The next section of code extends from line 22 to 39, beginning with the label LOOP. This code gets the prompt, displays it on the screen, then reads the user's response from the keyboard. Then it checks the first two characters in the user's response to see if they are NO. The WRITE program does this by loading the immediate value NO and comparing this to the first word (2 bytes, 2 characters) of the buffer. If the two match, control goes to the next sequential instruction, which jumps to the BYE sequence and returns to the CLI. If the first word in the buffer isn't the characters NO, the entire line is written to the terminal (@CONSOLE) and the disk file, and the sequence that begins with LOOP is repeated.

The error/return sequence starts on line 42 and extends to line 48. If there is an error condition, the program branches to ERROR, which puts the appropriate flags in AC2 (line 42), then branches over the SUB instruction to return (with ?RETURN) to the CLI. The flags tell the CLI to interpret the code in AC0 and display an explanatory message. For a normal return, after a person has typed NO↓, the program branches straight to BYE (line 47). BYE zeros AC2 to indicate a normal return, then makes a return (?RETURN) to the CLI.

Line 52 starts the prompt, which is the text string "Again?" followed by a NEW LINE character (<12>).

The parameter packets and buffers follow. Most parameter packet symbols need one word. MASM allots only one word to user symbols with the .ENABLE WORD statement on line 57.

Turning to page 2, you'll see the first parameter packet, CLIMS on lines 1 through 8, that's needed by the ?GTMES call. We used symbol ?GARG in the first offset because we wanted to get an argument. We put the argument number, 1, in the next offset, CLIMS+?GNUM. (Argument number 0 would be the program name, which we don't want here.) The next offset, CLIMS+?GRES, has a byte pointer, NAME*2, to the space that will receive the first argument. The .DWORD pseudo-op instructs MASM to give this symbol two words of storage. The .LOC on line 8 establishes the end of the packet and protects locations not described within the packet. End each packet with .LOC pkt-addr+pkt-len.

On line 10, NAME will receive the filename (pathname) from the ?GTMES call. It has enough space for a pathname of 64 characters.

Starting on line 14 and extending through line 27, is the I/O packet for the terminal (CON). Offset CON+?ISTI determines a change in record format if needed, the use of data-sensitive records, and the opening of the terminal for Input and Output (?OFIO). On line 17, CON+?IBAD, a two-word (.DWORD) byte pointer, points to the buffer BUFF. All user typed lines will be read into this buffer and written from it. Next, on lines 19 and 20, CON+?IRECL specifies a maximum record length of 120 characters. This is more than enough, since each record will be a line of text typed from the terminal keyboard. Next, CON+?IFNP has a two-word (.DWORD) byte pointer CON*2 to the area that holds the filename. The last two offsets contain –1. We don't want to specify anything, but must set these to –1. If we had omitted them, MASM would have set them to 0, which would have meant something else.

Following the CON packet, on lines 29 through 31, the program identifies the terminal's filename, @CONSOLE. @CONSOLE is the generic name for the terminal input and output files.

Next, on lines 35 through 53, is the I/O packet for FILE. Offset FILE+?ISTI stipulates changing format if needed, opening the file for appending, creating the file if it doesn't exist, creating data-sensitive records, and opening the file for input and output. In FILE+?IBAD, a two-word (.DWORD) byte pointer, points to the buffer BUFF. This is the same buffer that the terminal uses, so we don't have to worry about changing buffer byte pointers. FILE+?IRCL specifies the same record length as CON+?IRCL — 120. The value for FILE+?IRLR is 0; (actually we needn't have specified this location at all, but put it here because you might want to use it in later programs to check the number of bytes read or written.) As with CON, the last two offsets contain –1; otherwise, they are of no concern to us.

Line 57 reserves the buffer, BUFF, into which the program reads each line typed on the terminal keyboard. This buffer is 60 words or 120 bytes long.

Last, on line 58, we end (.END) the program, directing control back to WRITE.

(These packets may seem pretty tedious — but after you write one general-purpose packet, you can put it in a separate file and simply copy it into programs.)

The Errors

The final portion of the listing shows three errors: two on page 1, line 44; and one on page 2, line 58.

As with error reports from other utilities, you cannot assume that the precise number of assembly errors is the number of error codes you receive. Take the error codes to be general indicators of error conditions. Here, three errors were caused by two mistakes.

The first mistake is a typing error on page 1, line 47 — the incorrect `BY:` instead of the correct `BYE:`. The second mistake on page 2, line 58 involves the use of the symbol WRITE. WRITE should have been the name for the start of the program, not .WRITE. Deleting the period from the beginning of .WRITE should clear up this error.

Using SED or SPEED, fix the two erroneous lines. Change line 47 from

```
BY:     WSUB    2, 2    ; Set for ...
```

to

```
BYE:    WSUB    2, 2    ; Set for ...
```

Change line 58 from

```
.END    .WRITE          ; Start at the beginning.
```

to

```
.END    WRITE           ; Start at the beginning.
```

WRITE.SR with No Assembly Errors

Now for reassembling the WRITE program. Type

```
) xeq masm write ↓
```

MASM reports no errors, so we can continue to link WRITE:

```
) xeq link write ↓
OPTIONS: LINK  REVISION  nnnn  ON  date AT  time
WRITE.PR CREATED
```

Like MASM, the Link utility reports no errors. (Occasionally, Link reports *OVERWRITE PREVIOUS* errors in assembly language programs. This usually means mistakes in the parameter packets, causing the Link utility to overwrite a location already assigned a value. This kind of mistake can result from carelessness with .DWORD. If you get such an error, check your packets — especially your use of .DWORD.)

Will the WRITE program run correctly? Even though there are no assembly or link errors, there may be some errors of logic or design in the program. So run it by typing

```
) xeq write ⤸
```

```
*ERROR*
NO SUCH ARGUMENT
ERROR: FROM PROGRAM
xeq,write
```

This error occurred because there is no output filename. (The ?GTMES call couldn't find argument number 1.) Try again, using the output filename RECORD:

```
) xeq write record ⤸
```

```
Again?
```

Looks good — this is the WRITE program prompt. So type something:

```
something ⤸
something
Again?
```

Looks even better. The program wrote back our line and displayed the prompt again. We don't know whether it wrote to disk file RECORD yet. Try another line:

```
something  else ⤸
something  else
Again?
```

To check the disk file, RECORD, we need to get back to the CLI. Type the terminating sequence, NO:

```
NO ⤸
NO
Again?
```

That NO should have terminated WRITE and returned control to the CLI. But WRITE is still running. Try again:

```
NO ⤸
NO
Again?
```

Looks like a bug. To get back to the CLI, press

```
CTRL-C  CTRL-B

*ABORT*
CONSOLE INTERRUPT
ERROR: FROM PROGRAM
xeq,write,record
```

CTRL-C CTRL-B returned control to the CLI. Now, what about file RECORD?

```
) type record ⤸
something
something else
NO
NO
```

The disk file logic seems fine. Aside from the NO terminator problem, the WRITE program seems to be in pretty good shape. Check by running it again:

```
) xeq write record ⤸

Again?

What about the terminator problem? ⤸
What about the terminator problem?
Again?
NO ⤸
NO
Again?
```

Clearly, the terminator problem isn't going to go away. Interrupt WRITE and check RECORD again from the CLI:

```
CTRL-C  CTRL-B

*ABORT*
  .
  .
  .

) ty record ⤸
something
something else
NO
NO
What about the terminator problem?
NO
```

Everything works except the NO terminator, which should take us back to the CLI. The next step is a very common one in assembly language programming: debugging.

Introducing the AOS/VS Debugger

This section gives an overview of the debugger. Read it before you operate the debugger, as described in the next main section, "Debugging the WRITE Program."

The AOS/VS Debugger is a system utility that lets you examine and make minor modifications in your program as you run it.

Most debug commands employ the ESC key, which, as in SPEED, echoes as $. Here are the debugger commands you'll be using:

Command	What it Does
[addr] /	Opens *addr* as a one-word examine, and displays contents. Establishes one-word examines until you change this.
[addr] \	Opens *addr* as a two-word examine, and displays contents. Establishes two-word examines until you change this.
	For either *addr* command, the *addr* can be symbolic, numeric, or both — e.g., WRITE+2. Without *addr*, the symbols / or \ direct the debugger to display the contents of the current address.
Function Keys 3,6,7	These function keys control the format of the debugger display; we describe them in "Display Formats."
CR or ^	Pressing the CR key directs the debugger to open and display the *next* location. A caret (SHIFT-6) tells the debugger to open and display the *previous* location.
[ac]$A	Displays contents of accumulator *ac*. An ac can be 0, 1, 2 or 3. Without an ac, the $A command directs the debugger to display all accumulators.
[addr]$B	Sets a breakpoint (at which execution will stop) at address *addr*. Without *addr*, $B displays all breakpoints.
[cmd] $H	Provides help. If you omit *cmd*, the debugger will display a general Help message; if you include *cmd*, the Help facility will describe the command.
$?	Explain a *?* error message. The debugger error message is *?* and $? tells it to explain.
$P	Runs the program from the current breakpoint.
[addr]$R	Runs the program from its beginning, or from *addr*. You must type $R initially; then you can use $P, or *addr*$R.
$Z	Leave the debugger and return to the CLI.

When you display addresses, the debugger treats the CR and NEW LINE characters differently. A CR opens and displays the next location; NEW LINE simply closes the current location (if open) and displays nothing. (If your terminal lacks a CR key, use the RETURN key; if it lacks both CR and RETURN, use the LINE FEED key.)

To debug programs, the debugger uses a program symbol table file, named program-name.ST. The Link utility automatically creates this file along with the program file, and you will see it in your directory.

When you execute the debugger, it announces itself and prints its prompt. The prompt is an underscore (_); the debugger prints its prompt whenever it is ready to accept a command. You must type symbol names in uppercase, but debugger commands can be in either upper- or lowercase.

If you type something it doesn't understand, it will display a question mark (?); type $? (ESC ?) for a description. Generally, its error handling is sound, so you needn't worry about making mistakes.

Display Formats

The debugger always displays *locations* symbolically from the beginning of the program. The symbol it uses is the last location entered (.ENT) in the program. If no location was entered (.ENT), it uses ?USTA for the beginning of an unshared program. In the WRITE program, we entered (.ENT) on the symbol WRITE, so it will display user locations relative to WRITE.

When you start it, the debugger displays location *contents* as numeric octal values. This is called format or mode 1.

To see the displayed value in a different format, use function keys, the row of keys above the main keyboard on your terminal. The debugger documentation includes templates that fit over these keys and describe the format they produce; but you may not have the template, so we'll describe the keys here. Function key 1 (F1) is the leftmost key, and the keys' numbers are counted from left to right.

Function key 1 displays in numeric format. This is the default so you'll rarely need to press F1. The F3 (the third) key displays location contents in instruction format — e.g., `WBR  WRITE+100`. You'll use F3 a lot. The F6 key displays in byte pointer format, which is handy when you want to track down a location using the byte pointer. The F7 key displays in values in ASCII (e.g., the `NO ⤵` in the WRITE program), and you'll be using this fairly often. The F10 key allows you to examine a number as an error code.

For any location or accumulator, only one debugger format gives a useful picture of the contents. If the contents are an instruction, like XWLDA or WBR, the appropriate function key is F3. (An assembler listing can help in this process; for example, the listing in Figure 14-3 tells you that `WRITE+25` contains an instruction.)

If the contents are a byte pointer, like `PROMPT*2`, use the F6 key to display the location pointed to. If the contents are a value, as with many parameter packet offsets, the default mode is fine. If the contents are ASCII characters, such as Ag, the appropriate key is F7.

Usually, you can tell which display mode is correct because other modes will give surprising results. In practice, you may just press all the function keys (1 through 10) sequentially until you see something reasonable. This is the great thing about function keys — they're fast.

Debugger Breakpoints

A *breakpoint* is a location where you want the debugger to stop your program during execution. This gives you a chance to examine the accumulators and locations within your program. You may set as many breakpoints as you choose. The first breakpoint is number 0, the second number 1, and so on.

Each breakpoint address you specify can be a symbolic expression or a number. Symbolic expressions are easier. You can use each location you entered (.ENT) in a program as a symbolic address: e.g., `LOOP+3` if you entered LOOP. By entering (.ENT) WRITE in the WRITE program, you name a symbolic location for use in debugger expressions.

As mentioned before, the start of each unshared program is known to the debugger as ?USTA. Thus, even if you do not enter the symbol WRITE, you could have used ?USTA. Still, it's a good idea to enter at WRITE because it's familiar and because it's easier to type than ?USTA.

In any case, to set a breakpoint at the beginning of WRITE, issue the debugger command `WRITE$B`.

Do not set breakpoints within system calls. System calls are macros that the assembler expands into several words. You can define a breakpoint at the beginning of a system call or at one of its two returns, but not in the middle of the call. Also, don't set a breakpoint on the second word of a two-word instruction like XNLDA.

Examining and Changing Memory Locations

You can examine the contents of any location by typing the symbolic or numeric address of the location followed by a slash (for one word) or backslash (for two words). For example,

```
_ WRITE+24 / 111470_ ↓
_ WRITE+25 \  24416200146
```

WRITE+24 and WRITE+25 are Debug *addresses* or *addrs*. To open and display successively higher locations, press the CR key.

To display successively lower locations, use a caret (^) (SHIFT-6). In debugger dialogs, we show symbols as follows: the CR key as CR, the caret as ^, and the NEW LINE character as ↓. For example,

```
_ WRITE+24/       111470    _ CR

WRITE+25/       122071      _ ^

WRITE+24/       111470      _ ↓
```

As you can see, the debugger initially displays location contents as octal numbers, which don't mean very much.

The F3 key displays the contents as an instruction, and the F7 key displays the contents in ASCII. For example,

```
WRITE+25/    122071 _ F3    XLEFB 0,?USTA+174   _ F7    $9
```

The *XLEFB* is a two-word instruction that starts at WRITE+25. The F3 key shows the entire instruction, and F7 shows the first two bytes (*$9*), as appropriate for ASCII. Note that function key F3 interprets both words of the instruction even though you used a single-word examine (/). This is a convenience.

To change the contents of a memory location, open and display its contents, then type the new contents and press NEW LINE. The new contents can be an octal number, or it can be an instruction like WMOV. If you type in a one-word instruction, the current location will be changed; if you type a two-word instruction, the current and next locations will be changed; and if you type a three-word instruction, the current and two following locations will be changed.

Suppose you found that the two locations starting at WRITE+300 contained zero, but you wanted them to contain WMOV 0,1. The following sequence displays the locations, changes them, and verifies the change:

```
_ WRITE+300 \    000000000000 _    WMOV 0,1 ↓

_      .\       21336200000    F3       WMOV 0,1 _
```

Changing the contents of an accumulator is even easier. Open the accumulator with the n$A command; then type the new contents. For example, to place the value 1604 in AC3 and verify it,

```
_ 3$A    nnnnnnnnnnnn    _    1604 ↓

_ 3$A     000000001604   _
```

Starting or Continuing to Run Your Program

The first time you run a program in a debugging session, type **$R**.

When you want to continue from a breakpoint, type **$P**.

Initially, R is set to the address specified by the .END pseudo-op in the assembly language source program. Sometimes you may want to run the program from a different address. You can do this with the addr$R command.

Ending a Debugging Session

When you have finished debugging a program and want to return to the CLI, simply type

```
$Z
```

The changes you have made to your program during the debugging session do not become part of the program disk file when you leave the debugger. To make permanent changes, you must go through the cycle of editing your source program, assembling it, and linking it. Another way is to use the system disk File Editor (FED) utility to change the program file on disk. This is less desirable because the source program retains its errors. FED commands are identical to the debugger's, but it cannot set breakpoints or run a program.

Debugging the WRITE Program

When we left the WRITE program, we had a program with no assembly errors. It even ran properly, except that when we typed the NO terminator, it didn't return to the CLI.

The debugger can help find out why WRITE doesn't terminate on NO.

To start debugging the program, type

```
) debug write record ↓
```

The debugger identifies itself and displays its banner, then the initial contents of the accumulators:

```
AOS/VS  USER DEBUGGER, REV n
000000000000 0000000000 000000000000 0000000000 00...
-
```

The five numbers indicate the contents of the four accumulators and the carry.

Getting Help

Before getting into the WRITE program, let's see what help is available. Type ESC H:

```
$H       Welcome to DEBUG, the assembly language ...
```

To get information about the A command, try

```
A$H        -EXAMINE/MODIFY ...
```

The Help facility can be very useful, especially when you've forgotten command syntax.

Getting into the WRITE Program

We're ready to start debugging, using the MASM program listing as a guide. First, think about the problem: all system calls and I/O in the program work, but the compare logic on page 1, lines 30 through 34 doesn't work properly. This code should return control to the CLI when the user types NO, but it doesn't do so.

For practice, look around the symbol WRITE. This symbol starts a system call, which the debugger will show as a sequence of seemingly irrelevant instructions (beginning with LLEF,) followed by the WBR error return. First, press the ALPHA LOCK key if your terminal has one so the symbols you type will match the ones stored in the symbol table. Now type

```
_ WRITE/       131751    _ F3     ^^ LLEF 2,WRITE+116,1 _    CR

WRITE+1/ 000000 _
```

A system call starts with LLEF, a three-word instruction, followed by XJSR, a two-word instuction, followed by the system call word. So press CR 5 times to get to WBR:

```
CR

WRITE+2/  000115 _     CR

WRITE+3/ 143031 _          CR

WRITE+4/ 100006 _      CR

WRITE+5/  000307 _     CR

WRITE+6/ 115270    _ F3      WBR WRITE+100   _    CR
```

As you can see, there's a lot to a system call. (If you're simply looking around, single-word examines, with the slash /, will usually do the job.)

Let's take a look around NLDAI (location 53 on the listing), because that's where the problem is:

```
_ WRITE+53 \    30612247117    _  F3      NLDAI 47117 0 _
```

The NLDAI looks okay. Press

```
CR

WRITE+55\ 25302200172   _  F3      XWLDA 1,WRITE+251,1_   )
```

The wide load after the NLDAI looks okay. Now, check the prompt, which begins at location 111:

```
_ WRITE+111 \       10131660551    _  F7     Agai _    CR

WRITE+113 \ 15617605000 _      F7       ^^ n?<12><0>_    CR

WRITE+114\ 00000600001    _ )
```

The prompt looks okay. The F7 key shows that it has *Agai* in the first two words and *n?<12><0>* in the second two words. *<12>* is a NEW LINE character; *<0>* is a null.

Setting Breakpoints

Now, we're ready to set a few breakpoints. We will set two. The first will be on XWLDA at location WRITE+55 (to check NLDAI). The second will be on WSNE at WRITE+57 (to check XWLDA before a comparison of the two accumulators).

```
_ WRITE+55 \      n   _    F3    XWLDA 1,WRITE+251,1 _    .$B

_ WRITE+57 /      n   _    F3    WSNE 1,0_    ^.$B
```

Here, we opened each location and checked its contents before setting the breakpoint with the .$B command. Remember that the . represents the current location, so it works as a debugger address.

Having set two breakpoints, let's run the program:

```
_ $R      Again?
```

The WRITE program has run and displayed the prompt. The program needs something to read. So type

```
TEST1111 ↓

OB    WRITE+55

00000047117   37777777777   16000000654   06000000660 0...
```

This is the first breakpoint, just before the XWLDA instruction. It looks as if the NLDAI worked. Let's see what it loaded into AC0. Type

```
_ 0$A      00000047117   _    F7    <0><0>NO_    ↓
```

AC0 did get loaded with NO. AC1 has some residual data from the ?GTMES call; anyway, disregard this because nothing has been done with AC1 yet. AC2 has 653, which must be the CON packet address from the ?READ call. Check out 653 and the packet:

```
__ 653/          000004 __ CR          First location in packet.

WRITE+164/ 040032 __ CR                This is the ?ISTI word in the CON packet.
```

Let's look at the specified packet location (there are a number of locations not specified in this CON packet). From the listing, we know that .LOC CON+?IBAD is a two-word byte pointer to the filename. This location is at WRITE+167. Try using F6 for the byte pointer format. Type

```
_ WRITE+167\      34000001700_    F6    16000000740    0_ ↓

_ 740\     12421251524_    F7    TEST_    CR

WRITE+252\ 06114230461_    F7    ^^  1111_    CR

WRITE+254\ 01200..._    F7    ^^  <12><0><0><0>  _ ↓
```

Key F6 provided the number 740 (disregarding the leading 16 and separate 0). Checking 740, you'll find it starts the string TEST, 1111, and <12><0><0><0>. TEST1111↓ was what was just typed; thus 740 is the beginning of the buffer. So, in fact, ?CON+?IBAD does contain a byte pointer to the buffer, BUFF — and the byte pointer is working properly.

Continue with the program — which should take us to the WSNE.

```
_ $P

1B WRITE+57

00000047117 12421251524 16000000654 00000000000 0...
```

AC0 still has NO (...47117), but AC1 is loaded with something new. What?

```
_ 1$A        12421251524_     F7     ^^ TEST_   ↓
```

AC1 is loaded with TEST — the first four characters in the buffer. Continue. The program will loop back to type the prompt and read again. This time type NO, to try to terminate the program.

```
_ $P       Again?

NO ↓

0B WRITE+55

00000047117 12421251524 16000000654 00000000000 0...
```

You typed NO and the program continued to the first breakpoint. But AC1 hasn't been loaded yet. To continue, type

```
_ $P

1B WRITE+57

00000047117 11623605124 16000000654 00000000000 0...
```

AC1 has been loaded, but with what? It should be identical to AC0 now — but obviously isn't. Check both ACs:

```
_ 0$A        00000047117_     F7     <0><0>NO_   ↓

_ 1$A        11623605124_     F7     NO<12>T_    ↓
```

Now it's clear why the program won't terminate. After you type NO, AC0 has <null><null>NO, whereas AC1 has NO<char><char>. Because they don't match, the program never returns to the CLI. But why is AC1 getting the two extra characters?

The Bug

The bug is the wide load instruction. In the program, wide load XWLDA picks up the first *four* characters in the buffer. It should have been narrow load XNLDA. You've got to watch the width of loads and stores — matching them to the demands of the situation.

The solution is easy when you know the problem. Let's check it by making the wide load into a narrow load. Type the following without the NEW LINE or CR:

```
_WRITE+55\      n  _    F3    XWLDA 1,WRITE+250
                   _ XNLDA 1,WRITE+250 )
```

Now verify the change:

```
_ WRITE+55 \    n    _    F3     XNLDA 1,?USTA+251,1_   )
```

Having changed the wide load to narrow, start over. When the prompt appears, type NO again:

```
_ $P        Again?

NO )

OB WRITE+55

00000047117 11623605164 16000000653 06000000660 0...
```

The load hasn't worked yet. Try again:

```
_ $P

1B WRITE+57

00000047117 00000047117 16000000653 06000000660 0...
```

AC0 and AC1 now match! Verify the match as follows:

```
_ 0$A        00000047117   _   ^ F7     <0><0>NQ_   )

_ 1$A        00000047117   _     F7     <0><0>NQ_   )
```

And we can be certain that another Proceed command will get us back to the CLI; type

```
_ $P
)
```

If you check file RECORD with the TYPE command, you'll find that the test lines typed during the debugging process were appended to it.

Final Version of WRITE

Using SED or SPEED, change the XWLDA instruction to XNLDA on page 1, line 31 of the program, producing the line:

```
        XNLDA     1, BUFF        ; Get first word (2 bytes) of buffer.
```

Then leave the editor, reassemble, and relink WRITE:

```
) xeq masm/l=@lpt write ↓

) xeq link write ↓
.
.
.
```

Figure 14-4 shows the final version of the WRITE program, again without the cross-reference. Boxed areas indicate changes from the initial version.

```
SOURCE: WRITE            MASM 06.00.00.00       10-OCT-85 14:35:54  PAGE    1

01                              ;  This program gets a filename from the CLI XEQ
02                              ; command, opens the terminal and the specified file,
03                              ; writes a prompt to and reads a line from the
04                              ; terminal, and writes the line to terminal and file.
05                              ; To stop it, type NO and press NEW LINE.
06                              : XEQ WRITE <output-pathname>

07                                      .TITLE  WRITE
08                                      .ENT    WRITE     ; For debugging.
09                                      .NREL             ; Unshared partition.
10
11                              ; Get filename; open terminal and specified file.
12
13                              WRITE:  ?GTMES  CLIMS     ; Get the filename.
14 000006 UC 115270                     WBR     ERROR     ;   Error, process it.
15                                      ?OPEN   FILE      ; Open file, create if needed.
16 000015 UC 114370                     WBR     ERROR     ;   Error, process it.
17                                      ?OPEN   CON       ; Open terminal.
18 000024 UC 111470                     WBR     ERROR     ;   Error, process it.
19
20                              ; Write prompt, read line, check for NO, write files.
21
22 000025 UC 122071 000146      LOOP:   XLEFB   0, PROMPT*2  ; Get bptr to prompt.
23 000027 UC 121431 000137              XWSTA   0, CON+?IBAD ; Put in console pkt.
24                                      ?WRITE  CON          ; Write prompt to console.
25 000037 UC 110170                     WBR     ERROR        ;   Error, process it.
26 000040 UC 122071 000416              XLEFB   0, BUFF*2    ; Get bptr to I/O buffer.
27 000042 UC 121431 000124              XWSTA   0, CON+?IBAD ; Put in console pkt.
28                                      ?READ   CON          ; Read a line from console.
29 000052 UC 104670                     WBR     ERROR        ;   Error, process it.
30 000053 UC 143051 047117              NLDAI   'NO',  0     ; Put NO term. in AC0.
31 000055 UC 125451 000172              XNLDA   1, BUFF      ; Get first word (2
32                                                           ;  bytes) of buffer.
33 000057 UC 120611                     WSNE    1, 0         ; Skip if not equal NO.
34 000060 UC 104470                     WBR     GOODBYE      ;   User typed NO, exit.
35                                      ?WRITE  CON          ; Not NO. Echo line, go on.
36 000067 UC 101170                     WBR     ERROR        ;   Error, process it.
37                                      ?WRITE  FILE         ; Write line to file.
38 000076 UC 100270                     WBR     ERROR        ;   Error, process it.
39 000077 UC 164670                     WBR     LOOP         ; Do it all again.
40                              ;  Process error and/or return here.
41
42 000100 UC 153211             ERROR:  WLDAI   ?RFEC+?RFCF+?RFER, 2 ; Get err flgs.
43           00000150000
44 000103 UC 100270                     WBR     BYE                  ; Skip subtraction.
45
46 000104 UC 150531             GOODBYE: WSUB   2, 2                 ; Set for good return.
47                              BYE:    ?RETURN                      ; To CLI.
48 000110 UC 175070                     WBR     ERROR                ; ?RETURN error.
49
50                              ;  Prompt, terminated with NEW LINE.
51
52 000111 UC 040547 060551      PROMPT: .TXT  "Again?<12>"   ; Prompt.
53           067077 005000
```

Figure 14-4. Assembly Language WRITE Program without Errors (continues)

```
54
55                              ; Parameter packets and buffers.
56
57                                      .ENABLE  WORD ; Most pkt entries = 1 word.
58
59                              ; ?GTMES packet to get the message.




SOURCE: WRITE             MASM 06.00.00.00       10-OCT-85 14:35:54  PAGE    2

01 000115 UC  00000000006       CLIMS:  .BLK    ?GTLN           ; ?GTMES packet length.
02            00000000115  UC           .LOC    CLIMS+?GREQ     ; Offset to request type.
03 000115 UC 000003                     .WORD  ?GARG            ; We want to get an argument.
04            00000000116  UC           .LOC    CLIMS+?GNUM     ; Offset to arg NUM.
05 000116 UC 000001                     .WORD  1                        ; Arg number 1 is filename.
06            00000000121  UC           .LOC    CLIMS+?GRES     ; Offset to GTMES buf.
07 000121 UC  00000000246  UC           .DWORD  NAME*2          ; 2 wd bptr to NAME buffer.
08            00000000123  UC           .LOC    CLIMS+?GTLN     ; End of ?GTMES packet.
09
10 000123 UC  00000000040       NAME:   .BLK    32.             ; Reserve 64 chars for fname.
11
12                              ; Console I/O packet.
13
14 000163 UC  00000000030       CON:    .BLK    ?IBLT           ; Packet length.
15            00000000164  UC           .LOC    CON+?ISTI       ; Offset for statistics.
16 000164 UC 040032                     .WORD ?ICRF+?RTDS+?OFIO ; Change fmt+D-S rec+I&O.
17            00000000167  UC           .LOC    CON+?IBAD       ; Offset for buffer addr.
18 000167 UC  00000000520  UC           .DWORD  BUFF*2          ; 2 wd bptr to I/O buffer.
19            00000000172  UC           .LOC    CON+?IRCL       ; Offset for max rec len.
20 000172 UC 000170                     .WORD  120.             ; Max rec length of 120 chars.
21            00000000177  UC           .LOC    CON+?IFNP       ; Offset for file name ptr.
22 000177 UC  00000000426  UC           .DWORD  CNAME*2         ; 2 wd bptr to con filename.
23            00000000166  UC           .LOC    CON+?IMRS       ; Offset for memory size.
24 000166 UC 177777                     .WORD  -1               ; Memory block size, default.
25            00000000201  UC           .LOC    CON+?IDEL       ; Offset for delimter addr.
26 000201 UC  37777777777               .DWORD  -1              ; D-S del table addr; default.
27            00000000213  UC           .LOC    CON+?IBLT       ; End of terminal I/O packet.
28
29 000213 UC 040103 047516      CNAME:  .TXT "@CONSOLE"         ; Use generic name.
30           051517 046105
31           000000
32
33                              ; Open and I/O packet for FILE.
34
35 000220 UC  00000000030       FILE:   .BLK    ?IBLT           ; Packet length.
36            00000000221  UC           .LOC    FILE+?ISTI      ; Offset for statistics.
37 000221 UC 040272                     .WORD  ?ICRF+?APND+?OFCE+?RTDS+?OFIO
                                                                ; Change fmt+
38                                                              ; append+create+D-S recs+I&O.
39            00000000224  UC           .LOC    FILE+?IBAD      ; Offset for buffer addr.
40 000224 UC  00000000520  UC           .DWORD  BUFF*2          ; 2 wd bpt to I/O buffer
41                                                              ; (same as CON's).
```

Figure 14-4. Assembly Language WRITE Program without Errors (continued)

```
42            00000000227  UC          .LOC    FILE+?IRCL      ; Offset for max rec len.
43 000227 UC  000170                    .WORD   120.            ; Max record length of 120 chars.
44            00000000230  UC          .LOC    FILE+?IRLR      ; Offset for Ret. Len Rec.
45 000230 UC  000000                    .WORD   0               ; Sys returns # of chars transferred.
46            00000000234  UC          .LOC    FILE+?IFNP      ; Offset for file name ptr.
47 000234 UC  00000000246  UC          .DWORD  NAME*2          ; 2 wrd bptr to NAME,
48                                                            ; obtained by ?GTMES.
49            00000000223  UC          .LOC    FILE+?IMRS      ; Offset for memory size.
50 000223 UC  177777                    .WORD   -1              ; Memory block size, default.
51            00000000236  UC          .LOC    FILE+?IDEL      ; Offset to delimter addr.
52 000236 UC  37777777777               .DWORD  -1              ; D-S del table addr, default.
53            00000000250  UC          .LOC    FILE+?IBLT      ; End of file packet.
54                                 ; I/O Buffer. At end to avoid WBR addr errors.
55
56 000250 UC  00000000074       BUFF:  .BLK 60.                ; Buffer of 60. words, 120. chars.
57
58                                      .END   WRITE           ; Start at the beginning.
```

Figure 14-4. Assembly Language WRITE Program without Errors (concluded)

Running the WRITE Program

The WRITE program should run properly now; we've spent enough time on it. Try it out:

```
) xeq write record ↓
```

```
Again?
```

```
Will the terminator work after all this? ↓
Will the terminator work after all this?
```

```
Again?
```

Now for the test: will WRITE terminate?

```
NO ↓
```

```
)
```

The terminator works — the program is correct.

You can use the TYPE command to see that RECORD contains all messages from the session. The WRITE program is correct, and all ventures into the text editor, MASM, Link, and the debugger paid off. WRITE is a neat little program, and you can use it, or parts of it, to produce log files and do other things.

Summary

Having debugged the WRITE program, you have completed the assembly language minicourse given in Chapters 13 and 14. You've read about the Macroassembler, symbols, instructions, pseudo-ops, words, bits, system calls, parameter packets, the WRITE program itself, and the AOS/VS Debugger.

It's been a lot of work, but you've acquired a sound basic knowledge of Data General assembly language programming.

What Next?

You can start writing your own programs, using the manuals described in Chapter 16, or turn to Chapter 15 to learn about the Sort/Merge utility. You can return to an earlier chapter and review a text editor or program development in a high-level language. See Chapter 16 for details on all pertinent user documentation.

End of Chapter

Chapter 15
The Sort/Merge Utility

The Sort/Merge utility is a programming tool that you can use to change the order of records (parts of files), combine several files into one, or even translate EBCDIC files into ASCII files.

These are the steps you follow to sort a file with the Sort/Merge utility:

1. Select the data file to be sorted. It might be a payroll record, a customer list, or any kind of data file produced by an application program. If you have no existing file, then create the file with the SED or SPEED text editor:

   ```
   ) xeq sed filename ↓
   ```

   or

   ```
   ) xeq speed/d filename ↓
   ```

   With your chosen editor, type in the records that make up the file. (This chapter explains records and the types of records that AOS/VS supports.)

2. Then, with SED or SPEED, create a command file. This file passes directives to the Sort/Merge utility. Type

   ```
   ) xeq sed command-filename ↓
   ```

   or

   ```
   ) xeq speed/d command-filename ↓
   ```

3. Then execute the Sort/Merge utility with the command,

   ```
   ) sort/c=command-filename ↓
   ```

   If the output file already exists, add the /O switch.

4. When the sort completes, check the error file and the sorted file with the CLI TYPE command. If you receive an error message or see something incorrect, return to step 1 to correct the input file, or to step 2 to correct the command file. If there are no errors, go to step 5.

5. You're done! You can use Sort/Merge to reorganize your own files.

Record Formats

The term *record* pertains to all computer programming — not just to the Sort/Merge utility. Records are clusters of information (bytes) that a program either reads from, or writes to a hardware device, such as a terminal or disk. When a program reads a record, what it reads is called input; when a program writes a record, what it writes is called output. The read/write operations together are called record input/output, or record I/O.

Text editors, your own computer programs, and the Sort/Merge utility all read and write data in record units. A text editor reads lines of text from the terminal keyboard (input), and then writes the text to the terminal screen (output). For the editor, each text line is a record: it sees each line as a series of bytes ended with a special character, NEW LINE.

Computer systems can use several different formats to identify records. AOS/VS recognizes four of these formats, each having the following characteristics:

Record Type	Format
Data Sensitive	Each record has a terminator (also called a delimiter), which is commonly a NEW LINE (12_8), CR (Carriage Return) (15_8), form feed (page break or 14_8), or null (000). The system treats all characters preceding the delimiter as a record. The SED text editor as well as many Data General compilers require files with a data-sensitive delimiter at the end of each line (record).
Fixed Length	Each record has a constant length. The system treats each group of this length as a record, and doesn't look for a delimiter. Fixed records are useful for information that adapts well to a constant length, such as telephone numbers, card images, and databases for commercial applications.
Variable Length	Each record often differs in length from others in a file, and so a header, which is part of each record, identifies the size of the entire record.
Dynamic	There is no inherent format for dynamic records: neither header information nor a delimiter. Instead, the variables in a program work with "raw data," directing the system to read or write a given number of bytes. AOS/VS uses dynamic format for its logging of system activity.

Figure 15-1 shows how AOS/VS stores each record type.

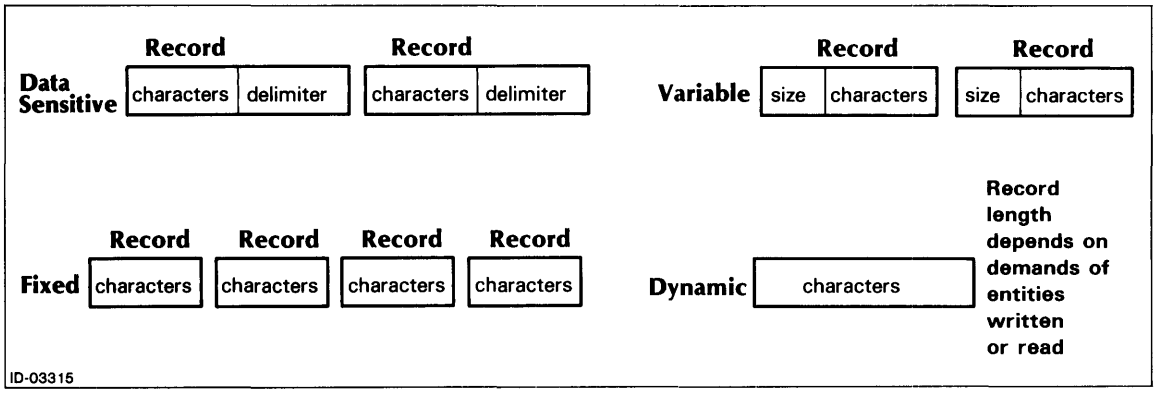


Figure 15-1. AOS/VS Record Formats

As a rule, the creating program determines the structure and length of a record, and a program usually specifies record format when it *opens* a file. The program Open statement sets up a channel between the computer and the file through which records can flow.

Although AOS/VS recognizes four types of records, the text editors and compilers that run under AOS/VS most often require data-sensitive records. The Sort/Merge utility, which we describe next, works with three record types: data sensitive, fixed length, and variable length records; it doesn't work with dynamic formats. The Sort/Merge utility can convert variable length records to ones of fixed length, and work with a file of any size.

Record Fields

Fields are character positions within records. For example, the record shown in Figure 15-2 has five fields, identifying a name, address, city, state, and phone number. Normally, other records in the same file would also have five fields, with identical starting and ending character positions. This allows all the records to be read (and the fields changed as needed) consistently. It also allows the records to be sorted by any of the five fields. For example, one could organize records with the name field (positions 1 through 20) first, or the state field (positions 51 through 53) first, or organize records with the name field sorted within the state field. The Sort/Merge utility will reorganize records in ascending or descending order of numbers or letters.

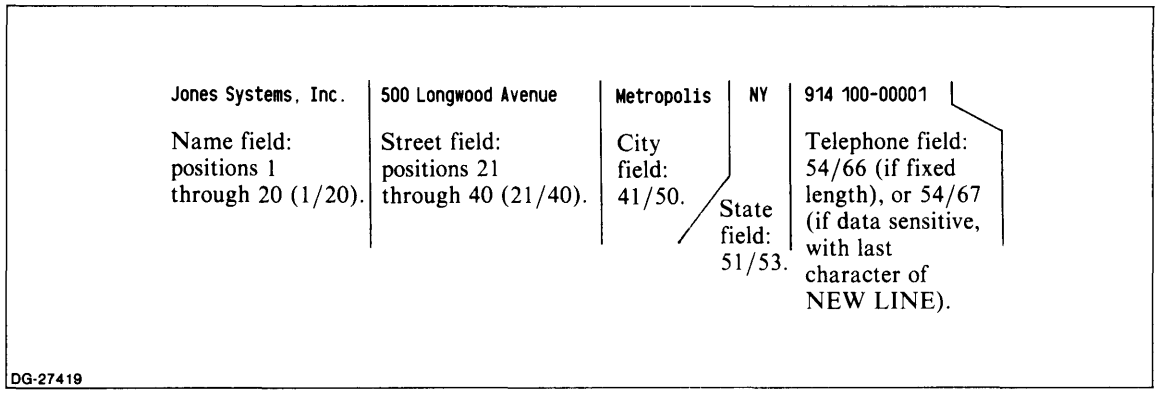


Figure 15-2. Anatomy of a Record

Creating the Input File

The sample file that we'll have the Sort/Merge utility reorganize is a simple list of consultants. Each record consists of five fields: all fields occupy the same character positions as shown in Figure 15-2; and all records end with a NEW LINE, making them data sensitive.

If you want to try the Sort/Merge utility, we suggest that you first create a directory for it. For example, type

```
) dir/i ↓
) cre/dir sort ↓
) dir sort ↓
```

Now, execute the SED or SPEED text editor, and create the input file which we'll call CONSULTANTS. Type

```
) xeq sed consultants ↓
```

or

```
) xeq speed/d consultants ↓
```

Now type in the records that constitute the file CONSULTANTS, as shown in Figure 15-3. The fields are identical to those defined in Figure 15-2.

```
Erpf, Rosemary      311 East 83rd       New York   NY 212 744-8991
Blatt, Tom          152 Newton Street   Weston     MA 617 893-3111
Houple, Howard      10 Torr Road        Andover    MA 617 475-7719
Fonseca, Agnes      1114 Martin Street  Laurel     VT 802 498-7775
Power, Maureen      43 May Street       Cambridge  MA 617 661-9693
Ewell, John         83 Summit Street    Portland   ME 207 767-4193
```

Figure 15-3. CONSULTANTS: The Input File to Sort/Merge

Note that the file CONSULTANTS consists of six data-sensitive records. The name field occupies positions 1 through 20, regardless of the length of the name; the street field occupies all characters in positions 21 through 40, and so on. The records hold a total of 67 characters, the last one being a NEW LINE character.

Writing the Command File

Having created an input file, you now need a command file that will point to the input file and pass directives to the Sort/Merge utility.

Execute the text editor again, and create a command file called NAME_SORT. Type

```
) xeq sed name_sort ↵
```

or

```
) xeq speed/d name_sort ↵
```

Enter the commands shown in Figure 15-4 into the file NAME_SORT.

The first line of NAME_SORT identifies the input file, CONSULTANTS and its record format. The second line supplies the name of the output file, LAST_NAME_SORT, which the utility will create when it writes the sorted records to a file. And the line following names the field to be sorted, or the criteria by which the reorganization should be based. In our example, we'll organize the records by the name field, position 1 through 20. By default, the utility sorts in ascending order — A to Z, 1 to 10. The file then declares the action the utility is to perform — sort, not merge. It closes with an end statement. Notice that each line in the command file ends with a period, and both filenames are in quotation marks.

```
INPUT FILE IS "CONSULTANTS", RECORDS ARE DATA SENSITIVE UPTO 67 CHARACTERS.
OUTPUT FILE IS "LAST_NAME_SORT".
KEY 1/20.
SORT.
END.
```

Figure 15-4. NAME_SORT: The Command File for Sort/Merge

Running Sort/Merge

You now have an input file and a command file, so you can try reorganizing the file. Execute the Sort/Merge utility with the command

```
) sort/c=name_sort ⤸
```

```
AOS/VS Sort/Merge - Rev. n     date     time
.
.
.
```

The Sort/Merge utility displays its banner and begins the sort. It reads the input file, CONSULTANTS, and sorts the records according to the last name, and puts the sorted records into the file LAST_NAME_SORT. Finally, it displays a summary of the sort, noting the number of records sorted and other statistics, and then returns control to the CLI.

Type the output file to see what happened:

```
) ty last_name_sort ⤸
```

```
Blatt, Tom          152 Newton Street   Weston     MA 617 893-3111
Erpf, Rosemary      311 East 83rd       New York   NY 212 744-8991
Ewell, John         83 Summit Street    Portland   ME 207 767-4193
Fonseca, Agnes      1114 Martin Street  Laurel     VT 802 498-7775
Houple, Howard      10 Torr Road        Andover    MA 617 475-7719
Power, Maureen      43 May Street       Cambridge  MA 617 661-9693
```

Looks good! The names are now in ascending alphabetical order.

Many times you'll want to organize records according to several criteria. For example, with consultants lists it's useful to order records by state, with names listed alphabetically within each state. To add a state key, execute the SED or SPEED editor, and insert a new line 3: key 51/53. The command file now reads

```
INPUT FILE IS "CONSULTANTS", RECORDS ARE DATA SENSITIVE UPTO 67 CHARACTERS.
OUTPUT FILE IS "LAST_NAME_SORT".
KEY 51/53.
KEY 1/20.
SORT.
END.
```

The command file now directs Sort/Merge to reorganize records according to the state field, positions 51 through 53, and then on a second pass, to alphabetize the name field (within a state) positions 1 through 20. Try to execute Sort again by typing

```
) sort/c=name_sort ⤸
AOS/VS Sort/Merge - Rev. n      date     time
** /O is required to allow the overwrite of an existing file **

*ERROR*
ERROR(S) IN THE COMMAND FILE
.
.
.
```

However, this time you receive an error message, and the Sort program terminates because the original output file, LAST_NAME_SORT, still exists in the working directory. We either need to delete the original file or append an /O switch to the SORT command, directing Sort/Merge to overwrite the existing output file. Add the /O switch, typing

```
) sort/c=name_sort/o ↵
```

This time the program completes without any error messages. Look at the new file:

```
) ty last_name_sort ↵


Blatt, Tom          152 Newton Street   Weston     MA 617 893-3111
Houple, Howard      10 Torr Road        Andover    MA 617 475-7719
Power, Maureen      43 May Street       Cambridge  MA 617 661-9693
Ewell, John         83 Summit Street    Portland   ME 207 767-4193
Erpf, Rosemary      311 East 83rd       New York   NY 212 744-8991
Fonseca, Agnes      1114 Martin Street  Laurel     VT 802 498-7775
```

Now our file CONSULTANTS lists people alphabetically by state.

Using Sort/Merge for Program Conversion

Another application for the Sort/Merge utility is character set translation. For example, assume that you have an EBCDIC file that you want to use under AOS/VS, which uses ASCII files. The file contains fixed-length 80-character records in 50-record blocks (in IBM parlance, LRECL=80, BLKSIZ=4000). You want the Sort/Merge utility to translate the file from EBCDIC to ASCII, and then insert a NEW LINE character to delimit each record.

The input file for the program conversion is a tape file, so your first step is to create a command file of instructions. Execute a text editor and create a command file named CONVERT that holds the following commands:

```
INPUT FILE IS "@MTB0:0", RECORDS ARE 80 CHARACTERS,
BLOCKS ARE 4000 CHARACTERS.
OUTPUT FILE IS "MY_CONVERTED_FILE".
TRANSLATE 1/LAST USING EBCDIC_TO_ASCII.
INSERT "<012>" AFTER LAST.
COPY.
END.
```

Then execute the Sort/Merge utility with the command,

```
) sort/c=convert ↵
```

Sort/Merge reads from the first file, which is file 0 of the tape on drive @MTB0. It produces an output file named MY_CONVERTED_FILE. You can modify the output file with a text editor. If it's a program file, you can compile the program. In cases where the disk file contains more than one source program, you can use a text editor to copy each source program into its own disk file.

What Next?

If you want to experiment with a language, or review material on the CLI or a text editor, turn to the appropriate chapter.

For more details on the Sort/Merge utility see the *Sort/Merge with Report Writer User's Manual (AOS and AOS/VS)* and the *Sort/Merge Utility User's Handbook (AOS)*, a pocket summary. For AOS/VS documentation in general, see Chapter 16.

End of Chapter

Chapter 16
Documentation Guide

This chapter describes the documentation Data General provides for products available with AOS/VS systems. It has three main sections:

- *Reading paths for AOS/VS users.* We chart product areas you may investigate, depending on your role as a general user, an applications or system programmer, or a system manager/operator.
- *An annotated bibliography*, describing documentation available for AOS/VS products. It's organized according to the product areas depicted in the reading paths shown in Figures 16-1 through 16-4.
- *Ordering information*, directions for obtaining documents from Data General.

Reading Paths

The material that you will want to read about any product depends on your role. We envision four types of readers:

- *A general user*, anyone who wants to use the AOS/VS system to perform a given task.
- *An applications programmer*, someone who writes programs that handle an organization's needs.
- *A systems programmer*, a person who writes programs that primarily assist system development: compilers, data management systems, operating systems.
- *A system manager/operator*, a person who administers system software, generates the operating system, decides who can use the system and what privileges they will have. A manager/operator often decides what programs will run and when they will run; he or she also plans data backup, recovery, and restart procedures. In many cases, this person operates or oversees the operation of system hardware — the tape drives, printers, and disk drives.

Reading paths for each of these readers follow.

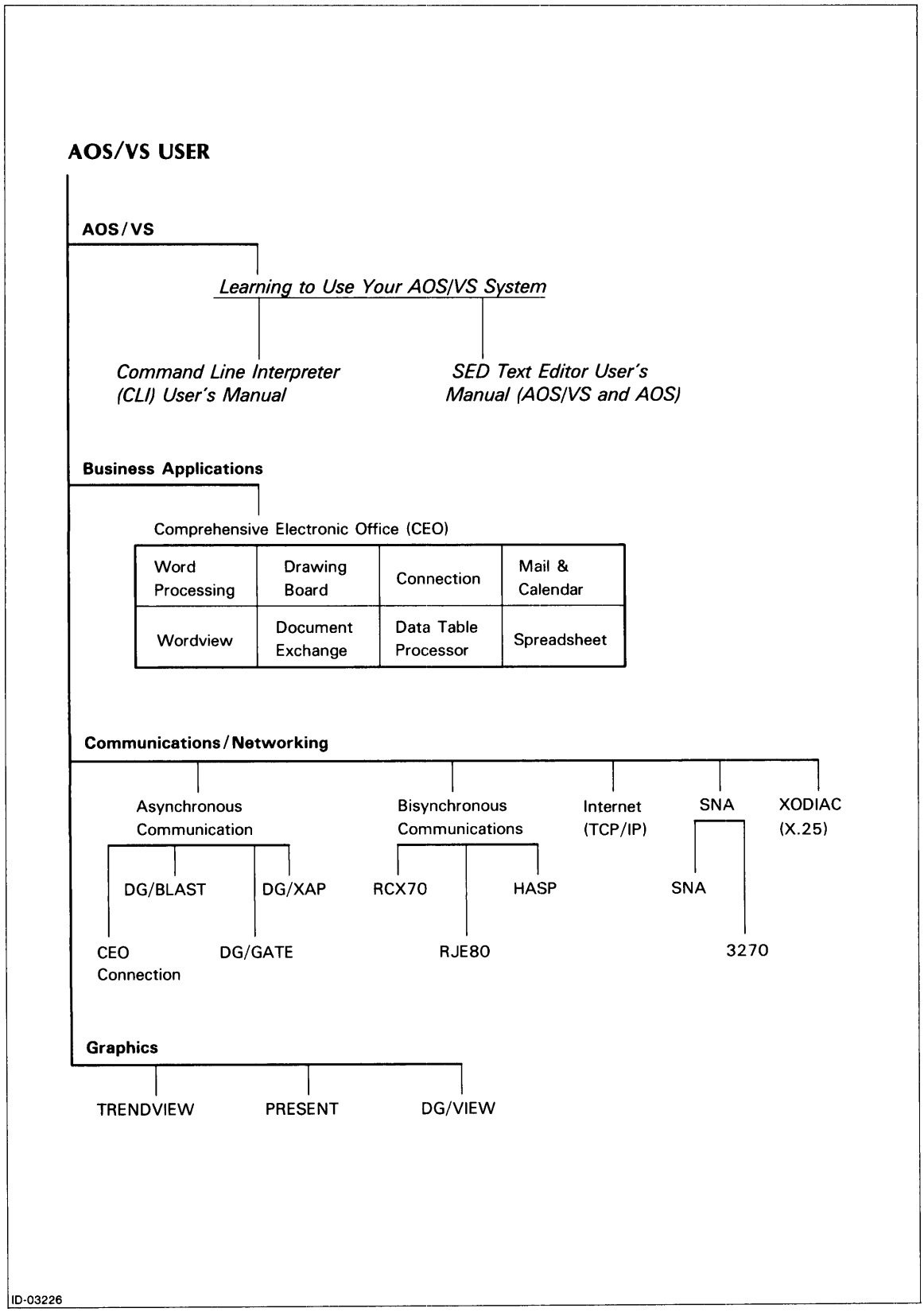


Figure 16-1. An AOS/VS User's Reading Path

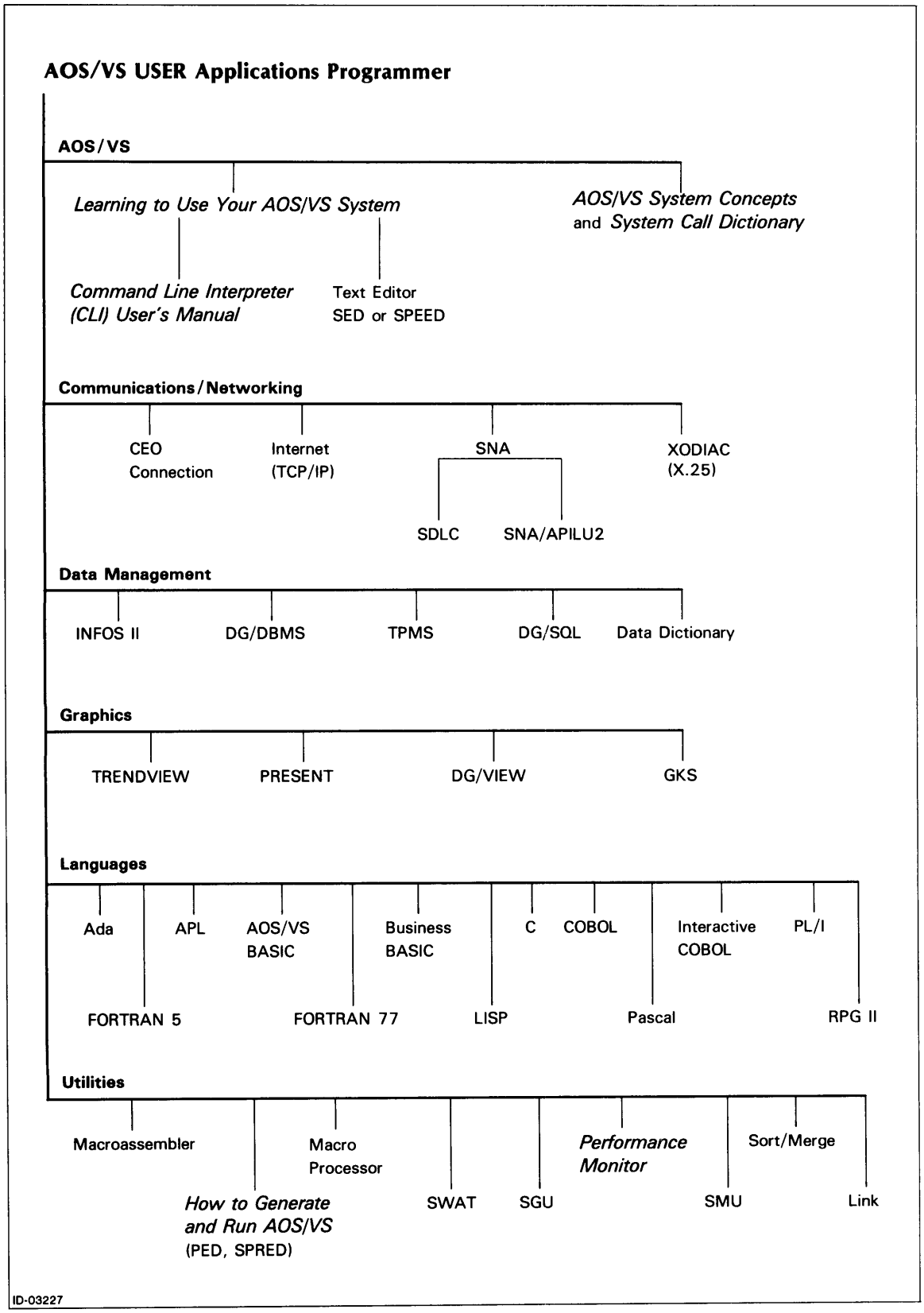


Figure 16-2. An AOS/VS Applications Programmer's Reading Path

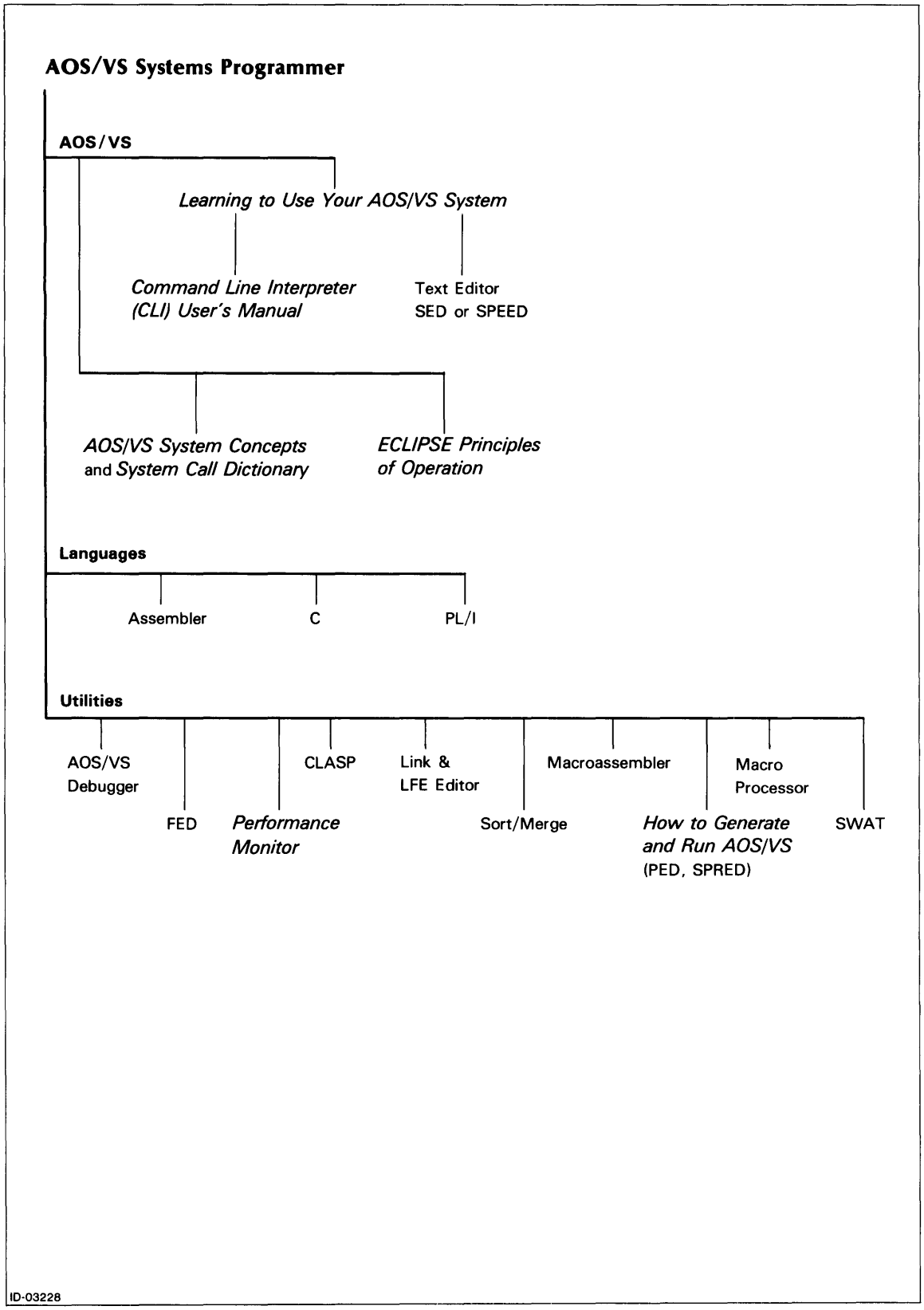


Figure 16-3. An AOS/VS Systems Programmer's Reading Path

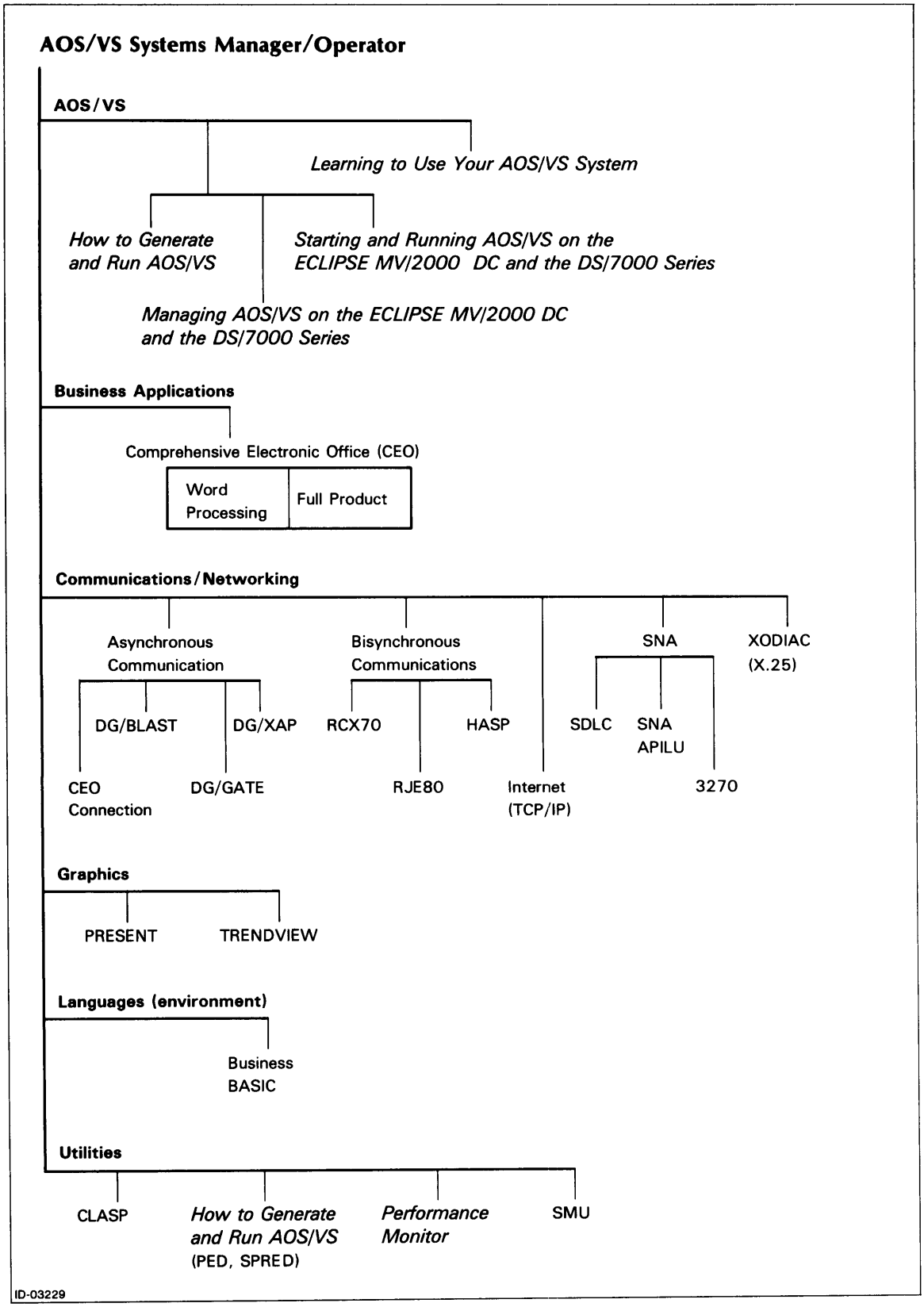


Figure 16-4. An AOS/VS Systems Manager/Operator's Reading Path

Bibliography

The annotated bibliography which follows describes documentation available on AOS/VS products. It's organized alphabetically by product area: AOS/VS, Business Applications, Communications/Networking, Data Management, Graphics, Languages, and Utilities, as reflected in Figures 16-1 through 16-4. Following the bibliography, you will find an explanation of how to obtain the books that you need.

AOS/VS

AOS and AOS/VS User's Handbook, 093-000150.
A short form of the CLI manual, summarizing CLI commands as well as the SED, SPEED, and AOS/VS Debugger commands. This is a handy reference for all users.

AOS/VS Commands, Macros, Programs, and EXEC Commands, 069-000108.
A reference card, most useful for system managers and operators.

AOS/VS System Concepts, 093-000335.
An explanation of basic AOS/VS concepts and how families of system calls work together. It's intended for system programmers, although application programmers, writing assembly language subroutines, will find it useful. It's a companion manual to the *System Call Dictionary*, described below.

Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS), 093-000122.
A guide to the CLI, the AOS/VS system command language, for all users. The manual describes how to invoke system utilities, execute user programs, maintain files, and write CLI macros. It includes a dictionary of commands and utilities.

ECLIPSE® 32-bit Principles of Operation, 014-000704.
A survey, for the assembly language programmer, of the 32-bit architecture of the ECLIPSE MV/Family computers: its addressing, data protection system, data types, instruction set, and I/O. It ships with machine-specific supplements:

- *DS/7700 System Principles of Operation,* 014-001217.
- *ECLIPSE MV/2000™ DC and DS/7500 Systems Principles of Operation,* 014-001203.
- *ECLIPSE MV/4000® System Principles of Operation,* 014-001226.
- *ECLIPSE MV/8000® Principles of Operation,* 014-001227.
- *ECLIPSE MV/10000™ Class Principles of Operation,* 014-001228.
- *ECLIPSE MV/20000™ System Principles of Operation,* 014-001169.

A separate document exits for the MV/4000 SC and Data General DS Systems, listed below.

ECLIPSE MV/4000® SC and Data General DS Systems Functional Characteristics, 014-001066.
A survey, written for the assembly language programmer, of the 32-bit architecture of the ECLIPSE MV/4000 SC and Data General DS system computers — their addressing, data protection system, data types, instruction set, and I/O. It's an essential guide for the assembly language programmer.

How to Generate and Run AOS/VS, 093-000243.
A clearly written description of how to build and run an efficient AOS/VS operating system. Topics include how to tailor an operating system for your particular hardware and software, how to build and maintain the multiuser environment, how to manage system resources, and how to keep your system secure. It details common procedures such as startup, daily operation, shutdown, formatting disks, and file backup and restoration. It's an essential manual for the system manager/operator, and programmers will find that some utilities explained here (PED and SPRED in particular) are helpful in program development.

Learning to Use Your AOS/VS System, 069-000031.
This book.

Managing AOS/VS on the ECLIPSE MV/2000™ DC and the DS/7000-Series Systems.
An interactive tutorial, distributed with AOS/VS software, geared to the novice system manager/operator for an ECLIPSE MV/2000 DC or Data General DS/7500 system. It explains how to start, run, and shut down AOS/VS. The tutorial discusses trouble-shooting, backing up and restoring files, optimizing performance, and modifying the system.

SED Text Editor User's Manual (AOS/VS and AOS), 093-000249.
An introduction to the line-oriented text editor, SED, that allows you to create and modify text files or program source code. It makes use of function keys, facilitating editing. Templates are available for the DASHER D2 series terminals (093-000248), D200 series terminals (093-000256), and D210 series terminals (093-000361). All system users will want to learn SED, SPEED, or the CEO Word Processor to write documents or source code.

SPEED Text Editor (AOS and AOS/VS) User's Manual, 093-000197.
A description of SPEED, a character-oriented text editor that allows you to create and modify text. SPEED has powerful macro capabilities and is virtually a text processing language. All programmers will find it a powerful programming tool.

Starting and Running AOS/VS on the ECLIPSE MV/2000™ DC and the DS/7000-Series Systems. 069-000129.
A description of the easy startup procedure and the menu-driven System Management Interface (SMI) program for these systems. Of primary interest to the system manager/operator, the manual is also suitable for the regular system user, containing information about the AOS/VS system, the on-line Help facility, and such tasks as file backup and recovery,

Starting Up and Shutting Down AOS/VS, 069-000107.
A reference card for the system manager, summarizing normal startup and shutdown, abnormal shutdown, and how to run ESD and FIXUP.

System Call Dictionary (AOS/VS and AOS/DVS), 093-000241.
A system call dictionary delineating the function, use, and implementation of each system call. It specifies accumulator input and output values, parameter packets, and error codes. It's a companion to *AOS/VS System Concepts* (above), and geared to the system programmer. An application programmer will find it useful on occasion — primarily when a program must make explicit system calls. For example, suppose a FORTRAN 77 program has to create a directory and make it the working directory. The program uses the ISYS function, as documented in the *FORTRAN 77 Environment Manual Manual (AOS/VS).* It also makes ?CREATE and ?DIR system calls, as documented in this *System Call Dictionary.*

Business Applications

Using the CEO Connection™ System, 069-000105.
Tells how to set up and use the CEO Connection, a product that makes it easy to communicate between personal computers and the AOS/VS CEO system. The manual tells how to use a personal computer as a CEO terminal and move files between CEO and a personal computer. Summaries provide technical information for programmers who are writing applications that address the terminal emulators that are an important part of the product.

CEO® Decision Base Spreadsheet Processor User's Manual, 093-000324.
An explanation of how to create and edit electronic spreadsheet documents within the CEO system.

The CEO® Document Exchange User's Manual, 093-000336.
A guide to the CEO Document Exchange I product that enables users to convert files from CEO word processing (WRD) format to Wang word processing format and vice versa.

CEO Drawing Board™ User's Manual, 069-700010.
Tells how to use the CEO Drawing Board graphics software to create, save, copy, modify, and print drawings, logos, charts, and graphs. It tells how to select colors and how to use a mouse or the cursor control pad to create drawings. The CEO Drawing Board on-line Help facility is an integral part of the CEO Drawing Board. Users who want more help can view an on-line tutorial that demonstrates each command in the CEO Drawing Board package.

CEO® Help Facility and the *CEO Electronic Manual.*
Both are integral parts of the CEO system software. A CEO user can get help anywhere in the CEO system with only two keystrokes. This help is context-sensitive; that is, it explains the user's current context and task. Users who want a complete explanation of a topic may read the on-line Electronic Manual. It provides up-to-date information on every aspect of CEO. When a user finishes reading any Help message or topic in the Electronic Manual, he or she returns precisely to the CEO task from which he or she asked for help.

The CEO® System Reference Card, 069-000033.
A pocket-size summary of the CEO System. It tells the CEO function of each key on the DASHER series of Data General terminals.

CEO Wordview™ Summary, 069-000106.
The CEO Wordview package allows you to create presentation-quality word charts. The summary tells how to select fonts and sizes and gives examples of each. It also tells how to print the resulting charts on graphics printers and plotters. The CEO Wordview Help Facility is on line and is always available to a user of CEO Wordview.

Getting Started with the CEO® System, 069-000036.
Gives a brief and simple introduction to the CEO system. It tells how to adjust your terminal and use your keyboard templates. It also explains how to create and file documents, schedule appointments, send and receive mail, and how to use the CEO system's on-line Help facility.

Managing CEO® Word Processing — Independent, 093-000271.
A manual for the person who will manage the CEO Word Processing — Independent software. The manual explains how to install and run the CEO Word Processor.

Managing the CEO® System, 093-000402.
Tells how to install and maintain CEO software. It provides specific installation instructions and explains how to create and maintain users' CEO profiles, to define printers, and to physically delete obsolete documents. It lists CEO error messages and tells how to respond to them.

PRESENT® Information Presentation Facility User's Manual, 093-000168.
A facility allowing you to select data from a sequential file, INFOS II file, or a CODASYL or relational database, and to format that data into the report or chart of your choice. With Data General TRENDVIEW (described below) graphics terminals and printers, PRESENT allows users to draw charts, graphs, and other illustrations. Templates are available for the DASHER D2 series terminals (093-000338), DASHER D200 series terminals (093-000339), and D210 series terminals (093-000357).

TRENDVIEW® Graphics Charting Package User's Manual, 069-700008.
An interpreter that works with Data General graphics terminals and printers, translating simple commands and numbers into images such as pie, bar, or line graphs. It can run interactively or with application programs. For all users.

Using the CEO® Decision Base Data Table Processor, 093-000391.
This manual describes how you can create and edit data table documents within the CEO system.

Using the CEO® Document Exchange II and VI Systems, 093-000411.
An explanation of how to set up and use two CEO products. CEO Document Exchange II enables CEO customers to extend their mailing capability by using Telex services. CEO Document Exchange VI enables CEO Customers to extend their mailing capability by using Teletex services.

Using CEO® Word Processing, 093-000285.
Tells how to use the CEO Word Processor, an integral part of CEO. The manual covers all aspects of CEO Word Processing, including special features such as list processing.

Using CEO® Word Processing — Independent, 093-000220.
Explains how to use the word processor that is independent of the full Comprehensive Electronic Office; it is the word processor without other CEO facilities, such as electronic mail or calendar.

Communications/Networking

Using the CEO Connection™ System, 069-000105.
Tells how to set up and use the CEO Connection, a product that makes it easy to communicate between personal computers and the AOS/VS CEO system. The manual tells how to use a personal computer as a CEO terminal and move files between CEO and a personal computer. Summaries provide technical information for programmers who are writing applications that address the terminal emulators that are an important part of the product.

DG/SNA APILU2 Reference Manual (AOS and AOS/VS), 093-000302.
A description of the APILU2 programmable interface to DG/SNA, of particular interest to applications programmers. The name APILU2 stands for the application program interface, Logical Unit type 2. APILU2 is a set of program calls that lets application programs running on Data General computers communicate to IBM hosts on System Network Architecture (SNA) networks. APILU2 is part of the DG/SNA product.

DG/SNA Operator's Guide (AOS and AOS/VS), 093-000283.
A description of Data General's SNA interface (DG/SNA) and the Synchronous Data Link Control program (DG/SDLC). The manual shows how to load, configure, start, and operate the DG/SNA and DG/SDLC software packages.

DG/SNA Reference Manual (AOS and AOS/VS), 093-000282.
An explanation — for application programmers — of Data General's Systems Network Architecture (SNA) emulator. DG/SNA provides an interface between a Data General AOS or AOS/VS system and an IBM SNA network. DG/SNA allows Data General terminals to emulate IBM-SNA devices; it also allows application programs to access the SNA network. It describes the IPC calls that are available to create a customized user interface to DG/SNA and an IBM SNA network.

Generating and Running DG/XAP™, 093-000352.
A guide — for the system manager — to DG/XAP, an asynchronous file transfer program for moving files between Data General Systems. The product is an economical alternative to XODIAC.

Generating, Running, and Using DG/GATE™, 093-000353.
A guide for system managers and users to DG/GATE, an asynchronous terminal emulator that's an economical alternative to XODIAC. The product also allows you to access public data networks such as Dow Jones and Dun & Bradstreet.

HASP Workstation Emulator User's Manual (AOS & AOS/VS), 093-000158.
Data General's HASP II Workstation Emulator Program, called HAMLET, emulates an International Business Machines (IBM) HASP workstation. HAMLET works with ECLIPSE computers and associated peripheral devices operating under Data General's AOS or AOS/VS operating system. The manual is written for all terminal users of HAMLET.

How to Use DG/BLAST, 069-100006.
A guide to DG/BLAST, an asynchronous file transfer program that provides an economical alternative to XODIAC. The program moves files over telephone lines to such systems as IBM, Hewlett-Packard, Apple, running BLAST™ or DG/BLAST software.

Managing and Operating AOS/VS Internet, 093-000400.
For network managers/operators, a description of how to create and maintain an AOS/VS Internet network. It describe the Network Generation Program (NETGEN), for configuring, generating, and changing a network; and the Network Operator Process (NETOP), for creating and controlling the Internet processes (TELNET, FTP, and TCP). Internet uses the TCP/IP protocol.

Managing and Operating the XODIAC™ Network Management System (AOS and AOS/VS), 093-000260.
A guide for network managers and operators, describing how to create and maintain a XODIAC network. It details the Network Generation Program (NETGEN), used for configuring, generating, and changing a network; the XODIAC Routing Analyzer (XRA), used for generating and updating routing tables for a large network (AOS/VS only); and the Network Operator Process (NETOP), used for creating and controlling the XODIAC processes (XTS, X25, RMA, SVTA, and FTA).

Programming with the XODIAC™ Network Management System (AOS and AOS/VS), 093-000175.
Presents a picture of the programming interfaces to the XODIAC Network Management System. A user can write an application program that uses the XODIAC transport service, which maintains a connection between two remote hosts. The XODIAC transport service is an implementation of the X.25 protocol. In addition, a programmer can use the services of the File Transfer Agent (FTA) and the Resource Management Agent (RMA).

RCX70 Reference Manual (AOS), 093-000172.
A guide to creating and maintaining an RCX70 terminal system. RCX70 emulates an IBM 3271 terminal cluster controller. The manual documents programmer and system reference material, as well as RCX70 system generation. Templates are available for the DASHER D200 terminal (093-000267) and D2 terminal (093-00017).

RCX70 Terminal Operator's Guide (AOS), 093-000170.
A tutorial handbook for clerical users of RCX70, a product that emulates an IBM 3271 terminal cluster controller. RCX70 provides communication between any Data General AOS or AOS/VS system and an IBM 360/370 host processor via a synchronous communication line.

Remote Job Entry Control Program (RJE80) User's Manual (AOS and AOS/VS), 093-000157.
An account of how to load, generate, and use the RJE80 IBM 2780/3780 terminal emulator. This emulator lets you transfer files between your AOS/VS system and a host computer at a remote site that supports 2780/3780 RJE terminals or has a Data General system running RJE80.

The SNA/3270 Operator's Guide (AOS and AOS/VS), 093-000287.
An account of how to load, configure, start, and operate the software in the SNA/3270 package. SNA/3270 allows Data General terminals to emulate IBM 3278 terminals, 3276 terminal controllers, and 3286 and 3289 printers.

SNA/3278/APL Operator's Guide, 093-701004.
SNA/3278/APL, when running on a Data General DASHER D450 or D460 display terminal, emulates an IBM 3278 terminal running APL (A Programming Language). This guide is for all users of SNA/3278/APL. Templates are available for the D450 terminal (093-000347) and the D460 terminal (093-000359).

SNA/RJE Operator's & User's Manual, 093-000301.
A complete guide to running and using the SNA/RJE product. Operators can learn how to load, configure, start, operate and shut down SNA/RJE. Users will learn how to send files and specify destinations for files over a Systems Network Architecture network to a Remote Job Entry subsystem.

Using AOS/VS Internet, 093-000399.
An account of the interactive Internet protocols: TELNET, which lets the user log on a remote system, and the File Transfer Protocol (FTP), which transfers files to or from a remote system. With these protocols, a user at a terminal on an AOS/VS host can communicate with a remote non-AOS/VS system that is running a compatible version of TELNET or FTP.

Using AOS/VS Transmission Control Protocol (TCP), 093-000398.
An explanation of the programming interface to the Transmission Control Protocol (TCP), which is part of the Internet family of protocols. AOS/VS TCP maintains a connection between an AOS/VS host and a remote non-AOS/VS host that is running a compatible version of TCP.

Using DG/3278, 093-000284.
A description for a Data General terminal user, on how to use the DG/3278 program to enter or read data, send messages, and communicate over an SNA network. DG/3278 belongs to a software package called SNA/3270. Templates are available for DASHER D2 terminals (093-000297) and D200 terminals (093-000298).

Using the XODIAC™ Network Management System (AOS and AOS/VS), 093-000178.
A guide to using the XODIAC Network Management System services from a terminal. The services documented are the Virtual Terminal Agent (VTA), which lets the user log on a remote system; the File Transfer Agent (FTA), which transfers files to or from a remote system; the Resource Management Agent (RMA), which accesses resources on a remote system; and X.29/Host Packet Assembler/Disassembler (X.29/Host PAD), which lets the user log on a remote AOS or AOS/VS host from a non-Data General terminal over a public data network.

Data Management

Data General/Database Management System (DG/DBMS) Reference Manual, 093-000163.
DG/DBMS is a database management system that lets you create, maintain, and use large network-type databases in interactive and batch environments. The DG/DBMS data description and data manipulation languages follow CODASYL 1978 specifications but include enhancements for transaction processing and ease of use. Templates are available for the DASHER D2 series terminals (093-000268), D200 series terminals (093-000269), and D210 series terminals (093-000356).

DG/SQL User's Manual, 093-701003.
The main reference on DG/SQL, a relational database management system that runs on Data General's ECLIPSE MV/Family computers under the AOS/VS operating system. The manual describes how to create and maintain a database, and how to write programs that access the database.

How to Use DG/Data Dictionary, 093-000408.
Written for the experienced application programmer, this collection of software tools helps you manage your data resources. The book describes how to operate the data dictionary and how it relates to other DG database products.

The INFOS® II Storybook, 069-000019.
A narrative about the INFOS system and how it helps a fictional corporation solve its data processing problems.

INFOS® II System User's Manual (AOS/VS), 093-000299.
INFOS II is a database-oriented file management system that lets users create, maintain, and use large databases in multiterminal or batch environments. It is a superset of an ISAM file system. You should have some experience with AOS or AOS/VS before using this manual.

INFOS® Query/Report Writer User's Manual (AOS), 093-000214.
A description of the Query and Report Writer facilities for Data General's INFOS system. Query lets you select and display INFOS records in an organized format without writing specific application programs. The Report Writer formats existing INFOS II records into a report form.

PL/I–INFOS® II Interface Reference Manual (AOS & AOS/VS), 093-000165.
Using extensive examples, this manual show you how to write PL/I programs that can manage files using the INFOS II file management system.

PRESENT® Information Presentation Facility User's Manual, 093-000168.
The PRESENT facility allows you to select data from a sequential file, INFOS II file, Data General/Database Management System database, or DG/SQL database, and format that data into the report or chart of your choice. With the Data General TRENDVIEW® program, graphics terminals, and printers, PRESENT allows users to draw charts, graphs, and other illustrations. Templates are available for the DASHER D2 series terminals (093-000338), D200 series terminals (093-000339), and D210 series terminals (093-000357).

Transaction Processing Management System (TPMS) Manager's Guide (AOS and AOS/VS), 093-000206.
Tells system operators or managers how to run TPMS: how to start up and configure the system, use its operator commands, run the user profile utilty, and define the data entry alphabet.

Transaction Processing Management System (TPMS) Reference Manual (AOS and AOS/VS), 093-000205.
A guide to designing and building screen-to-database transactions using TPMS, an on-line data entry system. TPMS controls terminals and screen displays (formats) quickly and efficiently. TPMS works with either of the Data General data management systems: INFOS II or DG/DBMS. COBOL or PL/I programs can call TPMS.

Graphics

CEO Drawing Board™ User's Manual, 069-700010.
Tells how to use the CEO Drawing Board graphics software to create, save, copy, modify, and print drawings, logos, charts, and graphs. It tells how to select colors and how to use a mouse or the cursor control pad to create drawings. The CEO Drawing Board on-line Help facility is an integral part of the CEO Drawing Board. Users who want more help can view an on-line tutorial that demonstrates each command in the CEO Drawing Board package.

DG/VIEW Summary, 069-000131.
Quick reference information about using a pointing device and working with DG/VIEW menus. This summary card also gives step-by-step instructions for starting programs and performing all the windowing operations.

Graphical Kernel System Reference Manual, 093-000355.
The Graphical Kernel System (GKS) is a standard specification of a graphics programming facility. This manual explains how to write, compile, and link graphics programs using GKS. It requires a good understanding of FORTRAN, but not necessarily of graphics programming.

Introduction to Computer Graphics, 014-001216.
An overview of the concepts of computer graphics and the key features of Data General's products, written for anyone with a basic knowledge of computer operation and programming. It includes a tutorial on computer graphics, discusses the Graphics Instruction Set (GIS), and describes such software tools as DG/GI and DG/GKS.

TRENDVIEW® Graphics Charting Package User's Manual, 069-700008.
The TRENDVIEW package is an interpreter that works with Data General's graphics terminals and printers, translating simple commands and numbers into images such as pie, bar, or line graphs. TRENDVIEW can run interactively or with application programs.

Using DG/VIEW, 069-000130.
An explanation of how to run programs in windows and use a pointing device (such as a mouse). The manual also shows the reader how to use all the windowing functions, including moving or sizing a window, scrolling a window's contents, and transferring data from one window to another.

Languages

Ada

Ada® Development Environment (ADE™) (AOS/VS) User's Manual, 093-000373.
The Ada Development Environment (ADE) is an integrated set of tools for software development and maintenance. The ADE manual is the companion document to the Ada programming language reference, described below, and assumes familiarity with the language.

Ada® Programming Language Reference Manual, 069-000073.
This is the American National Standard reference manual for the Ada programming language (ANSI/MIL-STD-1815A–1983), plus an addendum giving the implementation-specific details of Data General's version of Ada.

Ada® Source Code Debugger (AOS/VS) Reference Manual, 093-000380.
Instructs you on how to use the Debugger to monitor and control the execution of an Ada program, interactively tracking the program's execution and inspecting its variables. The Debugger works in conjunction with the Ada compiler and the Ada Development Environment (ADE); therefore you must also be conversant with that software and its documentation.

AOS/VS Ada® Programming Language and Ada Development Environment (ADE™) Handbook, 069-000075.
A condensed version of the Ada programming language reference and the ADE user's manual. In concise, reference form, it contains information about the syntax and semantics of the Ada programming language and about the ADE facilities and commands. It assumes familiarity with its two companion manuals about ADE.

APL

APL Reference (AOS/VS) Manual, 093-000274.
This manual describes APL (A Programming Language), an interpretive language developed during the 1960s for mathematical and analytical applications. APL has its own character set, which extends the ASCII set, its own editor, and an interpreter. Terminal character labels are available (093-000343), as well as an editor template for the DASHER D200 series terminal (093-000304).

Assembler

ECLIPSE® 32-bit Principles of Operation, 014-000704.
A survey, for the assembly language programmer, of the 32-bit architecture of the ECLIPSE MV/Family computers: its addressing, data protection system, data types, instruction set, and I/O. It ships with a machine-specific supplement:

- *DS/7700 System Principles of Operation,* 014-001217.
- *ECLIPSE MV/2000™ DC and DS/7500 Systems Principles of Operation,* 014-001203.
- *ECLIPSE MV/4000® System Principles of Operation,* 014-001226.
- *ECLIPSE MV/8000® Principles of Operation,* 014-001227.
- *ECLIPSE MV/10000™ Class Principles of Operation,* 014-001228.
- *ECLIPSE MV/20000™ System Principles of Operation,* 014-001169.

A separate document exits for the MV/4000 SC and Data General DS Systems, listed below. See also the *Macroassembler Reference Manual* listed under "Utilities."

ECLIPSE MV/4000® SC and Data General DS Systems Functional Characteristics, 014-001066.
A survey, written for the assembly language programmer, of the 32-bit architecture of the ECLIPSE MV/4000 SC and Data General DS system computers: their addressing, data protection system, data types, instruction set, and I/O. It's an essential guide for the assembly language programmer.

BASIC

AOS/VS BASIC Reference Manual (AOS/VS), 093-000252.
An introduction to AOS/VS BASIC and a description of its features. The manual gives the information needed to code, debug, and execute BASIC programs on the AOS/VS operating system. It also explains how to use the BASIC compiler and how to move Extended BASIC software to AOS/VS BASIC.

Business BASIC Reference Manual for Commands, Statements, and Functions, 093-000351.
An alphabetical dictionary of Business BASIC commands, statements, and functions. This is a reference manual for programmers, which supports revision 4.0 (or later) of AOS/VS Business BASIC.

Business BASIC Reference Manual for Subroutines, Utilities, and BASIC CLI, 093-000389.
An alphabetical dictionary of Business BASIC subroutines, utilities, and the BASIC CLI. The manual is intended to be used as a reference manual for programmers working with revision 4.0 (or later) of AOS/VS Business BASIC.

Business BASIC System Manager's Guide, 093-000388.
A guide to loading and generating Business BASIC. It is intended for the system manager/operator working with Revision 4.0 (or later) of Business BASIC.

Business BASIC Technical Concepts (AOS, AOS/VS, RDOS, DOS), 093-705004.
A presentation of technical information on file structures, INFOS, the File Maintenance utility, and the Screen Maintenance utility. It also contains a glossary and error message listing. The manual is a reference for programmers.

Extended BASIC User's Manual, 093-000065.
A straightforward explanation of Extended BASIC, which assumes that you have had some experience with the BASIC programming language.

A Guide to Using Business BASIC (AOS, AOS/VS, RDOS, DOS), 069-000028.
Introduces the reader to Business BASIC and describes its major features. It contains overview information on some Business BASIC utilities, such as the Database Generator (DBGEN), File Maintenance, and Screen Maintenance.

C

The C Language Summary, 069-000053.
A pocket guide, containing information that the C reference manual, described below, discusses in detail.

The C Language Reference and Runtime Manual, 093-000264.
A description of Data General's implementation of the C programming language and its associated runtime environment (library functions and macros) under the AOS/VS, AOS/RT32, MV/UX™, and DG/UX™ operating systems. This is a reference manual, not a tutorial. It assumes that the reader is an experienced programmer, already familiar with C, some other high-level language, and/or assembly language.

COBOL

COBOL Reference Manual (AOS/VS), 093-000289.
A description of all aspects of AOS/VS COBOL, including how to code, compile, debug, and execute COBOL programs. AOS/VS COBOL is based on the 1974 ANSI standard, with extensions added.

IC/EDIT: Interactive COBOL Editor, 055-000004.
A guide to the Interactive COBOL text editor, used to write interactive COBOL source code and documentation.

Interactive COBOL Programmer's Reference, 093-705013.
A manual, written for the experienced COBOL programmer, that defines the syntax and extensions of the Interactive COBOL programming language. In addition, it provides an overview of the language, describes file organization and access, and presents extended program examples.

Interactive COBOL User's Guide (AOS & AOS/VS), 069-705015.
A manual for Interactive COBOL programmers, describing the AOS/VS file system, system calls, runtime system, the compiler, and the debugger. Lists of error messages and their meanings are provided.

FORTRAN

Data General's FORTRANs, 069-000029.
An account of Data General's FORTRAN programming languages that compares them to each other and to the ANSI standards for FORTRAN. The manual briefly describes the earlier Data General FORTRAN languages and compares these to the ANSI 1966 standard. Then it compares FORTRAN 5 to the 1966 and 1978 standards, and finally compares FORTRAN 77 to the 1978 standard and to earlier FORTRAN languages.

FORTRAN 5 Reference Manual, 093-000085.
Describes the Data General FORTRAN 5, a superset of the ANSI 1966 FORTRAN that includes elements of IBM FORTRAN IV and UNIVAC FORTRAN 5.

FORTRAN 5 Programmer's Guide (AOS), 093-000154.
A guide with two parts: the first part details aspects of operating FORTRAN 5 under AOS and AOS/VS, of error handling, and of the runtime environment, and the second part explains the FORTRAN 5 runtime routines.

FORTRAN 77 Environment Manual (AOS/VS), 093-000288.
A compilation of the Data General extensions to ANSI standard FORTRAN 77. They include special purpose subroutines, multitasking, runtime interaction with the operating system, and an interface with Data General's database management system, DG/DBMS.

FORTRAN 77 Reference Manual, 093-000162.
A manual for FORTRAN programmers explaining the features and operation of Data General's FORTRAN 77, which includes the full ANSI 1978 standard plus many extensions. To help the user, the extensions are shaded. An appendix explains how FORTRAN 77 differs from FORTRAN 5. Another appendix explains the conversion of Digital Equipment Corporation VAX™ FORTRAN 77 programs and data files to their Data General MV/Family counterparts.

FORTRAN 77 Documentation Summary, 069-000080.
A summary booklet of the two FORTRAN 77 manuals described above.

LISP

Data General COMMON LISP Reference Manual, 093-701017.
The primary reference for the Data General COMMON LISP system that runs under AOS/VS and its interface to the MV/UX operating system. DG COMMON LISP comprises both a language system and an environment. The language system includes an interpreter, a compiler, and runtime libraries. The environment is a set of program development tools, including a debugger and the EMACS text editor.

Pascal

The Pascal (AOS/VS and DG/UX™) Language Summary, 069-000037.
A summary card of commands, syntax, data structures, reserved words, as well as command lines for compiling and linking.

Pascal (AOS/VS and DG/UX™) Reference Manual, 093-000290.
Intended for those experienced in high-level language programming, this manual describes the syntax and semantics of the Pascal programming language, as implemented by Data General. It tells you how to write, compile, link, and execute Pascal programs under AOS/VS or DG/UX.

PL/I

PL/I-INFOS II Interface User's Manual (AOS & AOS/VS), 093-000165.
Using extensive examples, this manual show you how to write PL/I programs that can manage files using the INFOS II file management system.

PL/I Reference Manual (AOS/VS), 093-000270.
A description of Data General's implementation of the PL/I programming language — with details on writing, compiling, linking, and running PL/I programs under AOS/VS. This PL/I implementation is compatible with the ANSI Standard PL/I General Purpose Subset (X3.74 – 1980) and has several extensions to the subset and to ANSI PL/I.

Plain PL/I (A PL/I Primer), 093-000216.
An introduction to Data General's PL/I programming language, an enhanced subset of ANSI PL/I. It is written to help programmers learn PL/I.

RPG II

Data General/RPG II Optimizing Compiler (DG/ROC) User's Manual (AOS & AOS/VS), 093-000279.
An explanation of the function of the RPG II Optimizing Compiler (DG/ROC) in the RPG II language system, focusing on the DG/ROC relationship to the RPG II Interpretive Compiler (DG/RIC). The manual shows how to code, compile, debug and execute RPG II programs using DG/ROC or, in special situations, DG/RIC.

RPG II Reference Manual, 093-000117.
A reference for the Data General's RPG II language system that assumes a knowledge of the RPG II language.

Utilities

Advanced Operating System/Virtual Storage (AOS/VS) Debugger and File Editor User's Manual, 093-000246.
A guide for programmers on how to use the AOS/VS Debugger (DEBUG) and File Editor (FED) utilities. These utilities can be used to debug assembly language programs and to edit AOS/VS disk files. Templates are available for the DASHER D2 series terminals (093-000276) and D200 series terminals (093-000278).

Advanced Operating System/Virtual Storage (AOS/VS) Link and Library File Editor (LFE) User's Manual, 093-000245.
A description of the Link and the Library File Editor (LFE). The Link utility builds object files into executable program files — it is a primary program development utility. Using the LFE utility you can create, edit, and analyze library files.

AOS/VS Performance Monitor User's Manual, 093-000364.
System managers, as well as system programmers, will find the Performance Monitor a helpful tool to optimize system or process performance. The package contains a monitor that shows what's happening while AOS/VS runs, a working set trace program that tracks activity in a process working set, and a tuning tool that allows you to change performance-related parameters in AOS/VS systems. Templates are available for the DASHER D2 series terminals (093-000371), D200 series terminals (093-000372), and D210 series terminals (093-000340).

Class Assignment and Scheduler Package, 093-000422.
The CLASP utility can help a system manager (and occasionally, a system programmer) tailor process scheduling in AOS/VS for the specific needs of a computer site. CLASP lets you create, monitor, and manage process classes and logical processors easily and quickly.

Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS), 093-000122.
A guide to the CLI, the AOS/VS system command language, for all users. The manual describes how to invoke such system utilities as FED, FILCOM, PED, SCOM, Sort/Merge, and SWAT.

How to Generate and Run AOS/VS, 093-000243.
Although designed for system managers/operators, this manual offers programmers valuable information on the PED and SPRED utilities. PED, the Process Environment Display utility, exhibits information on active processes, allowing you to observe working set size and CPU usage. SPRED, the Selective Preamble Editor utility, permits you to change values in the paging preamble and thus improve program performance.

Macro Processor for Procedural Languages (MPL) User's Manual (AOS), 093-000253.
A description of the Data General macro processor for procedural languages (MPL). MPL lets programmers define and call macros in programs written in high-level languages. MPL preprocesses source code by replacing macro calls with the statements contained in the corresponding macro definition.

SED Text Editor User's Manual (AOS/VS and AOS), 093-000249.
An introduction to the line-oriented text editor, SED, that allows you to create and modify text files or program source code. It has function keys, facilitating editing. Templates are available for the D2 series terminals (093-000248), the D200 series terminals (093-000256), and the D210 series terminals (093-000361). All system users will want to learn SED, SPEED, or the CEO word processor to write documents or source code.

SGU User's Manual (AOS and AOS/VS), 093-000305.
A guide for RPG II, TPMS, COBOL, and PROXI applications programmers, showing how to use the Screen Generator Utility to design screen menus and formats. Many templates are available for the following series of terminals:

- a COBOL template for the DASHER D2 (093-000118) and D200 (093-000051).
- an RPG II template for the D2 (093-000316) and D200 (093-000317).
- a TPMS template for the D2 (093-000318) and D200 (093-000319).
- an SLI template for the D2 (093-000344), D200 (093-000345), and D210 (093-000363).

Sort/Merge Utility User's Handbook (AOS), 093-000176.
This is a handy pocket reference of Sort/Merge directives and syntax.

Sort/Merge with Report Writer User's Manual (AOS), 093-000155.
Shows how to use the Sort/Merge utility under AOS/VS with or without the Report Writer. You should be familiar with the CLI, record I/O, and, possibly, a text editor to use this manual effectively.

Source Management Utilities User's Guide and Reference Manual, 069-40208.
SMU is a set of utilitiy programs that help manage files and develop software. It includes TCS, which is useful to anyone who's developing code and wants to store different versions of the same file. Both programmers and system managers will find the product helpful.

SPEED Text Editor (AOS and AOS/VS) User's Manual, 093-000197.
A description of SPEED, a character-oriented text editor that allows you to create and modify text. SPEED has powerful macro capabilities and is virtually a text processing language. All programmers will find it a powerful programming tool because it can edit such symbols as NEW LINE and CR characters.

SWAT® Debugger User's Manual, 093-000258.
The SWAT debugger an interactive symbolic debugger, works with C, COBOL, PL/I, FORTRAN 77, and Pascal programs. It lets you set breakpoints, display and change variables, and read lines from the original source programs.

Ordering Manuals

To order manuals, use the *Technical Information and Publications Service (TIPS)* catalog, which is available from your sales representative or by writing Data General directly:

TIPS
Educational Services — MS F019
Data General Corporation
4400 Computer Drive
Westboro, MA 01580

If you would like more of an overview of software products described in the preceding Bibliography, refer to *Data General/Information Systems for Commercial Applications,* 060-000030. This manual describes all of Data General's commercial products, support services, and the way they relate to each other. The *Data General/Information Systems* manual includes languages, data and terminal management, word processing, communications and networks, operating systems, documentation, and field support.

When ordering manuals through your Data General sales representative or from the TIPS catalog, give both the title and the part number to avoid confusing manuals with similar titles. The part number for a manual contains three fields:

- The first three digits identify a manual series.
- The next six digits identify a specific manual within a series. (If the three leading digits in this field are 0s, they are frequently omitted from publication listings.)
- The last two digits specify the revision of the manual. The original release is number 00, the first revision is 01, and so on. (Because these change fairly often, we omitted them from the manual numbers shown earlier. Your Data General representative will know what the latest revision is.)

For example, suppose that a manual has this part number:

093-000122-07

The 093 indicates a member of the licensed software series. The 000122 identifies a specific book (in this case, the *Command Line Interpreter (CLI) User's Manual).* The final numbers, 07, indicate that the book is in its seventh revision. You only have to supply 093-000122; Data General will send you the latest version.

Data General product users will read documentation from the following series:

010 *A configuration drawing.* Drawings in this series provide information for configuring and installing a system, subsystem, or option. This information may include any of the following: component specification, slot assignments, packaging, internal and external cabling, tailoring (including jumpers), rack mounting or installation, and options. You cannot acquire 010-series books unless you own the hardware they describe.

014 *A hardware reference.* Manuals in this series contain condensed hardware information. These manuals are often, but not always, complemented by more detailed manuals in the 015 series. The 014 manuals are primarily intended for those who use Data General equipment (as opposed to those who maintain it). Anyone interested may acquire 014-series books.

015 *A hardware technical manual.* Manuals in this series detail the hardware of Data General computer systems. The 015 manuals are generally intended for field service engineers who maintain Data General hardware products. You cannot obtain 015-series books unless you own the hardware they describe.

069 *An unlicensed software manual.* Manuals in this series are unlicensed books that contain introductory, general, or summary information. A new user or prospective customer would likely begin by reading 069-series books. These books are available to anyone interested.

085 *A Software Release Notice.* These notices are distributed on tape or diskette with a software release. A Release Notice may describe any or all of the following:

- Software part and model numbers.
- The organization of files on the release media, and what the files contain.
- Related manuals.
- Required hardware and software environments.
- Enhancements to the latest software release.
- Problems fixed in the latest software release.
- Software patches, corrections to a program that must be installed on disk.
- Documentation changes and additions.
- Information not available elsewhere — e.g., files of diagnostic messages or examples.

093 *A licensed software reference or user document, including special aids like function key templates.* Most software manuals belong to this series. Books in the 093-series are shipped with the product they describe. You cannot acquire 093-series books without agreeing to buy the pertinent Data General product.

Getting Updates

Data General continually improves its software products and updates its technical manuals. It sends the improved products and manuals to all customers who are members of the Software Subscription Service. A year's membership is included with the purchase of AOS/VS; membership in the subscription service is available thereafter.

When technical information changes, we update our manuals as soon as possible. Between manual revisions we distribute changes and corrections through Software Release Notices. A Release Notice accompanies every major software product release on magnetic tape or diskette; after loading it with the software, you can print it on your own line printer. Release Notice information is incorporated into later revisions of manuals.

What Next?

Having read about the manuals available with AOS/VS, you now have some perspective about Data General's entire product line for AOS/VS. Earlier chapters gave you the information you need to use an AOS/VS system.

You've really finished *Learning to Use Your AOS/VS System.* As you apply this knowledge in your own environment, you'll continue the learning experience. From time to time, you may want to refer to the earlier chapters or the Glossary to refresh your memory or to find out about other tools.

End of Chapter

Glossary

abort A process terminates as a result of a control sequence (such as CTRL-C CTRL-B) or a serious program error.

access control list (ACL) A list of privileges (which can include Owner, Write, Append, Read, and Execute access) associated with every directory and file. Each ACL specifies the type of access allowed for each user.

accumulator A hardware register within the processor, used for arithmetic, value comparisons, and address manipulations. An MV/Family processor can have eight accumulators accessible to users: four fixed-point 32-bit accumulators and 4 floating-point 64-bit accumulators.

ACL *See* access control list.

Ada A programming language designed for large scale and real-time systems, created during the 1970s in accordance with Department of Defense standards. Ada is named for Augusta Ada Byron (Lady Lovelace), the first programmer. She and Charles Babbage, the inventor of the calculator, studied probability and statistics — trying to beat the horses.

Advanced Operating System *See* AOS *or* AOS/VS.

ALC instruction An Arithmetic-Logical-Carry instruction, like WMOV or MOV, used in Assembly language for arithmetic, bit manipulation, and value comparison.

ANSI American National Standards Institute, a committee that publishes a wide range of standards for — among others — computer languages and tapes, machine screws, metric symbols and abbreviations, and electronic circuitry.

AOS Data General's Advanced Operating System for 16-bit ECLIPSE computers.

AOS/VS Data General's Advanced Operating System with Virtual Storage for 32-bit ECLIPSE computers.

archive *See* backup.

argument Something that is acted upon by a command, statement, or instruction. For example, in the command QPRINT MORTGAGE.F77, MORTGAGE.F77 is an argument to the QPRINT command.

ASCII American Standard Code for Information Interchange. This code establishes standard numeric values for each character used in text; the numbers range from 000 for the null character to 177_8 for the DEL character. An international character set extends the ASCII set with numbers from 200_8 to 377_8; these numbers indicate non-U.S., language-specific characters (for example, the United Kingdom currency symbol).

assembler A utility program that translates system calls and symbolic instructions into binary instructions for the computer.

assembly language A programming language that simpifies the process of producing machine language instructions for the computer. In general, each assembly language instruction corresponds to one machine language instruction, except that symbolic names represent numerical operation codes and addresses.

asynchronous line A communications line that has a variable time interval between successive bits, characters, or events. In an asynchronous protocol each character has its own "framing" information: traditionally one start bit and one stop bit. Terminals and intersystem communication lines generally use asynchronous lines. *See also* synchronous line.

background An RDOS division of main memory that allows two separate programs to run concurrently. RDOS might run in the background, while an application, such as a real-time processing program, might run in the foreground. AOS/VS is a multiprocessing system, which allocates memory in an entirely different manner.

backup The procedure of copying disk-based information to magnetic tape, diskettes, or disks for safekeeping; or the copied tapes, diskettes, or disks themselves.

bad block A flawed area on the magnetic surface of a diskette or disk. This area will not hold information.

BASIC The Beginner's All-purpose Symbolic Instruction Code, an easy-to-use interpreted language originally developed at Dartmouth College.

batch A technique of processing in a continuous, noninteractive stream. Batch jobs, submitted to the system with the QBATCH command, are processed by the system in one of four streams. Batch jobs do not require a terminal and execute without user intervention; they're suitable for large, well organized programs, such as large sorts.

baud The rate at which a line or modem can transfer data, expressed in bits per second (bps). On asynchronous lines each character typically requires 10 bits, so characters are transferred at one-tenth the baud rate.

bit A shortened form of the term binary digit. Each bit can assume one of two values, 0 or 1. In Data General products, 16 bits equal 1 word; therefore 1 word can represent 65,536 different numbers. Two computer words, or 32 bits, can represent over 4 billion numbers.

blocked process A process whose execution is suspended while it waits for an event to occur. A user process blocks by default when it creates another process.

bpi Bits per inch is a measure of data density on magnetic tape. Common standards are 800, 1600, and 6250 bits per inch.

break sequence A control sequence, which — when typed from a user terminal — can terminate binary mode or log a user off the system.

breakpoint A place where a debugger stops program execution. At a breakpoint, you can examine the current values of variables or, with assembly language, the values in accumulators.

browsing The act of looking through directories on the system — a practice that ACLs can substantially limit.

buffer A part of computer memory used to receive and temporarily store disk-based information.

Business BASIC An enhanced BASIC with ISAM capabilities. ISAM is an indexed sequential access method used by INFOS II.

byte One byte consists of 8 bits. Each byte is capable of storing one ASCII character, such as the letter A, or any number between 0 and 255.

C A general-purpose programming language that was developed with the UNIX™ operating system. It's used to write operating systems as well as major numerical, text-processing, and data-base programs.

CEO The Comprehensive Electronic Office, Data General's popular office automation product. It includes, among other facilities, electronic mail, a calendar, filing, and word processor.

checksum A test used to verify data integrity.

CLI The Command Line Interpreter for AOS/VS, a system utility program whose commands allow you to interactively maintain files, access all other utility programs, and do many other tasks. The CLI replaces the Job Control Language (JCL) facility in batch-oriented systems.

COBOL The Common Business Oriented Language, a very popular programming language for business that features English language constructions.

command In this book, a keyword that instructs the CLI, a text editor, or a debugger.

Command Line Interpreter *See* CLI.

compiler A utility program that translates the statements of a high-level programming language, such as FORTRAN or COBOL, into binary instructions for the computer.

console Another name for terminal. An interactive device with a keyboard for input and a screen or printer for output. The filename for a terminal is @CONSOLE. A *system* console is connected to the processor directly, whereas *user* consoles are indirectly connected to the processor through a multiplexor.

control character *See* CTRL key.

control key *See* CTRL key.

CPD Control Point Directory. A CPD is limited to the size specified at its creation.

CPU The Central Processing Unit or Processor, one of the three sections of a computer system — the others being Input/Output and Memory. The CPU decodes and executes program instructions, performs arithmetic and logical operations, and holds critical data. Another term for CPU is job processor.

CR The CR (carriage return) key acts as a line terminator, much like the NEW LINE key, except that CR truncates all characters to the right of the cursor before sending the line to the system.

CRT terminal A cathode ray tube or a video display terminal is an input/output device consisting of a keyboard and a television-like screen that displays characters.

CTRL key A CTRL (control) key is a terminal key that, by itself, does nothing, but with another character has special effects. The resulting "control sequences" are useful for screen editing and cursor or system output control.

cursor On a display terminal screen, the cursor indicates the current position on a line. It is either a box superimposed on a character position or an underscore beneath a character position.

data A general term describing any numbers, letters, or symbols that can be processed or produced by a computer.

data-sensitive record A cluster of bytes delimited by an agreed-upon character, such as the NEW LINE. *See also* record.

database A central location for data that can be shared by many users or programs.

debugger A utility that allows you to run another program, set breakpoints, stop execution at the breakpoints, and examine and change variables in the program. Debuggers can help you find program errors and understand the details of program execution. Data General has several debuggers, including SWAT™ for high-level languages, the AOS/VS Debugger, and the Interactive COBOL Debugger.

default; by default A value or parameter that a program uses unless you specify a different one. Two examples of default: the SED text editor displays line numbers by default; and when you open a disk file, it is opened for both input and output by default.

delimiter A special character that ends a data-sensitive record. Data General delimiters are ASCII NEW LINE (12_8), Form Feed (14_8), and Null (0).

device, device name A piece of hardware in a computer system — for example, a terminal, tape unit, or line printer.

directory A file whose sole function is to contain other files. Directories can help you organize and keep track of your files; the system itself uses them for this purpose. *See also* root directory, user directory, *and* UTIL.

disk A fast, mass storage device, consisting of one of more metal platters that rotate rapidly. Each platter has a magnetic coating that holds data. The operating system, all its directories and files, and all user directories and files are stored on disk.

disk block An area on a disk that contains 512 bytes of storage.

diskette A flexible disk used for software distribution and for file backup. Also called a floppy.

DISPLAY A utility that allows you to examine files with nonprinting characters (like program files) or display hexadecimal values in octal.

dump To copy information, usually from disk, onto a dump file. A dump file is often magnetic tape or diskette, but can be another disk or a disk file.

EBCDIC An abbreviation for expanded binary coded decimal interchange code. An 8-bit code — used by IBM and other manufacturers — to represent 256 unique letters, numbers, and special characters. Data General uses ASCII instead.

echo The system confirms characters by displaying them on a terminal screen.

editor, text *See* text editor.

emulator A program that enables a terminal or computer system to act like a specific (other) type of terminal or computer.

execute A directive to the CLI to interpret an instruction and perform the indicated operation(s).

extension *See* suffix.

field A set of one or more adjacent columns on a punched card; also one or more bit positions in a record consistently used to record similar information; also a division of the terminal screen according to function — for example, the command field and the text field in the SED text editor.

FILCOM A utility that compares files with nonprinting characters word by word (16-bit).

file A collection of information stored as a unit and identified by a filename.

file type A file's characteristics and function determine its file type. AOS/VS supports over 256 types of files, ranging from text files (TXT) to program files (PRV) to generic files (GFN).

filename AOS/VS filenames can contain from 1 to 31 of the following characters: letters, numbers, the underscore, the period, the dollar sign, and the question mark. Devices have standard filenames: @MTB0 refers to a tape unit; @LPT refers to a line printer queue; and @CONSOLE refers to a terminal. *See also* name.

firmware Instructions that control some aspect of computer hardware.

floppy A colloquialism for diskette. *See* diskette.

foreground An RDOS division of main memory. This method of dividing memory allows two separate programs to run concurrently; for example, a real-time process control program might run in the foreground, while a compiler, assembler, or payroll program might run at different times in the background. AOS/VS is a multiprocessing system, in which each process is a ground.

form feed A character, either CTRL-L or ASCII 14, that directs a printer to start a new page, or directs a terminal to clear the terminal screen.

FORTRAN Short for Formula Translator; it's one of the oldest and most popular high-level programming languages. Both the FORTRAN 5 and the FORTRAN 77 compilers run under AOS/VS.

FTA The XODIAC network's File Transfer Agent. It moves files efficiently from one computer system to another, and provides recovery in case of transfer failure.

function key One of the keys in the top row of a terminal keyboard; each function key represents a command, unique to the program you're running. A template, the plastic strip that fits over the keys, identifies the command/key relationship.

generic file A file that works as a pointer to a file identified at runtime; these generic files allow programs to be device independent. Program code can read and write to generic files, and then at runtime, CLI commands can link a generic file to an actual file — a device or a disk file. For example, the generic list file (called @LIST) can be set to @CONSOLE or to a disk filename, using the CLI LISTFILE command.

hard copy A printed paper copy of computer output, as opposed to a visual display or a magnetic tape copy; also a terminal that does not have a video screen, but prints its output on paper.

hardware Any of the devices that make up a computer system: the processor, user terminals, line printers, tape units, disks, etc.

hierarchy All processes are related in a structure that resembles an inverted tree. The highest processes are the system processes; user processes are subordinate to them. Use the CLI TREE command to identify process relationships.

high-level language A language — such as Pascal, FORTRAN 77, or COBOL — where each statement corresponds to several machine code instructions. Its counterpart is a low-level language, such as assembly language.

I/O Input/Output describes the process of reading information into computer main memory, or writing information from memory. The output can go to disk files, magnetic tape, card decks, the terminal screen, process I/O equipment, telephone lines, or microwave beams.

ICOBOL Interactive COBOL, a Data General COBOL that runs on all Data General computers.

input The computer process of reading information from a device (a disk, diskette, terminal keyboard, telephone line, microwave beam) into the computer's main memory.

interactive Descriptive of user input and system response dialog. You provide information and commands to the computer from the terminal keyboard, and the system responds by displaying information on the screen.

JCL Job Control Language, a language used to communicate with the operating system; the CLI is the JCL for AOS/VS.

job A printing or batch request made by a user.

job processor One of the three portions of a computer system — the others are Input/Output and Memory. The job processor decodes and executes program instructions, performs arithmetic and logical operations, and holds critical data. When a computer has one job processor, it's traditionally referred to as the CPU.

Kbyte An abbreviation for kilobyte. In main memory a Kbyte equals 1,024 bytes (or characters); on disk or diskette it equals 1,000 bytes.

keypad A group of keys on a terminal keyboard. Usually there are three: a main keypad, a cursor control keypad to its right, and a numeric keypad (used for data entry) on the far right.

library files (.LB) A collection of standard routines and subroutines, known as object modules, that the Link utility selects from the appropriate system library and builds into a user's compiled source file, wherever the code calls for an external subroutine. The result is an executable program file.

line A sequence of ASCII characters that ends with either a NEW LINE, form feed, or a null character; also, in communications, the physical medium of data transfer; *see also* synchronous line *and* asynchronous line.

line printer A high speed printing device that you access with the CLI QPRINT command or the queue name @LPT.

link The joining together of all necessary object modules to produce an executable program file. The AOS/VS Link utility (LINK.PR) performs this task.

link file A file whose sole function is to indicate the pathname of another file. For example, a link file MAR could call, or resolve to, a file named :UDD:CHRIS:MARCH_REPORT. See the CREATE command in Chapter 3 for details.

LISP Short for list processing. LISP is an interpretive language developed for manipulating symbolic strings and recursive data.

local Describes an item, such as a terminal or a CEO database, that is managed by your computer system — without a communications line to another computer system. The opposite of local is remote.

local host The computer to which your terminal is physically connected.

log on Sometimes called "log in." To pass a recognition procedure and be accepted by a computer system as a valid user.

macro A file containing a sequence of instructions or commands that can be executed by typing the filename. It may or may not require arguments. Macros are primarily timesavers, allowing you to write a series of directives only once, then invoke all of them by a single word.

magnetic tape A medium used for software distribution and data backup. Data General supplies four types of tape units: an MTB, an MTC, an MTD, and an MTJ. Tape unit device names are @MTxn, where x is type B, C, D or J, and n is the number on the unit thumbwheel (for MTB units) or selected during tape installation (for MTC, MTD, or MTJ units). Tape files are numbered sequentially from 0 to 99. With a tape on unit MTB0, you would access the first file as @MTB0:0, and the second file as @MTB0:1.

master directory An RDOS term referring to the directory where the RDOS system software resides. AOS/VS doesn't have a master directory.

Mbyte An abbreviation for megabyte. In computer memory a Mbyte is 1,048,576 characters; in disk storage, an Mbyte is one million bytes.

modem Short for modulator/demodulator. A device that connects a remote terminal to a computer over a telephone line.

mouse An input device that you move across a flat surface, sending signals to the computer.

MPL Data General's macro processor for procedural languages. MPL allows a programmer to define and reference macros for use in programs written in high-level langauges.

name AOS/VS filenames can be from 1 to 31 characters, including letters, numbers, the underscore (_), the period (.), the dollar sign ($), and the question mark (?). FORTRAN and AOS/VS BASIC symbolic entity names can be from 1 to 32 characters, including alphanumeric characters and the underscore. AOS/VS BASIC names must begin with a letter, whereas FORTRAN names can begin with a letter or a question mark. Business BASIC names must also begin with a letter but can contain only 6 alphanumeric characters. C names are primarily lowercase and contain alphanumeric characters, the underscore, dollar sign, and ampersand. COBOL and Interactive COBOL names can be from 1 to 32 uppercase letters, numbers, and the hyphen (-). Pascal names can have up to 32 characters, consisting of alphanumeric characters, the underscore, question mark, and dollar sign. Assembly language symbol names, by default, are unique to only eight characters and can include letters, numbers, the period, dollar sign and question mark. They must begin with a letter.

network A group of computer systems that can communicate via a communications link and that can share each others resources. The Data General XODIAC network system — with its X.25 protocol and agents FTA, RMA, and VTA — allows an AOS/VS system to participate in a Public Data Network or in local area networks. Data General also provides the Internet system, DG/BLAST, DG/GATE, and DG/XAP.

NEW LINE A character produced by the NEW LINE key, and represented frequently by the symbol ⤸. It is a delimiter, terminating command lines to the CLI and other programs, and delineating data-sensitive records.

object file Produced by a compiler or assembler, this is a file that contains binary translations of code from a source file. By default, the compiler or assembler assigns the same name to the object file that the user assigned to the source file, but it replaces the language-specific suffix of the source filename with the .OB suffix.

offset In assembly language, a location relative to another symbol (e.g., TABLE+1 or CON+?IBAD).

on line In direct communication with the computer. When a terminal is on line, the computer reads from the terminal keyboard and writes to its screen.

operating system A large program that manages and operates devices for users and user programs.

operator, system The person who physically operates a computer system for users. Sometimes, the system manager is the operator.

overlay A section of executable program code that the operating system can call into a reserved area of computer memory as needed. A program's overlays usually reside in an overlay file, identified by the suffix .OL. AOS/VS doesn't require the overlay mechanism, but it can be used by programs that run under both AOS and AOS/VS.

page In text editors, the number of lines between form feed (CTRL-L) characters (inserted in SED with the SPLIT command); also, in computer memory and disk I/O, 2048 bytes.

page fault An event in which the system must add a page to a process working set of pages because the process needs information that isn't in physical memory.

parent directory The directory immediately above a directory; for example, the parent directory of :UDD:ALEXIS is :UDD; and the parent of :UDD is the : (the root).

Pascal A high-level programming language designed for the development of source code that is both readable and easily maintained. Named after Blaise Pascal, French philosopher and mathematician, who was greatly admired by its originator, Niklaus Wirth.

password A code consisting of 3 to 15 AOS/VS filename characters. Your password in conjunction with your username identifies you as a valid user and allows you to log on the system.

pathname A route to a directory or individual file; for example, :UDD:ALEXIS:LEARNING:MACRO.CLI is a pathname from the root directory to the file MACRO.CLI. It establishes the path through all the directories leading to the file MACRO.CLI.

PED The Process Environment Display Utility. A program that displays information (i.e., the page fault rate, memory use, processor use) on active processes, so you can observe process performance.

peripherals directory The system directory that holds all files for all hardware devices, represented by the symbol @, as in @LPT.

physical address The address that directly refers to a hardware location in memory, as opposed to a logical address, which points indirectly to a hardware location.

PID Short for Process ID. The system assigns a PID number to each process at its creation.

PL/I Short for Programming Language 1, a programming language developed at IBM during the 1960s. It's the official Data General implementation language: many AOS/VS utility programs are written in it.

preamble Part of a program (.PR) file that specifies any of certain nonstandard parameters, such as the swap file size, to be used when the program runs. You can set its values with the SPRED utility.

primary partition An RDOS term referring to the area on disk that's available for system and user storage. There is no AOS/VS equivalent. See Chapter 1 for a description of AOS/VS file structure.

process Describes the way AOS/VS packages programming requests. A process consists of a program file, a work space in computer memory, and permission to use system resources.

program A series of instructions, translated into binary codes, that the computer can execute. Programs include the text editor that allows you to write the instructions, the compiler or assembler that translates them, and the Linker that positions them correctly. The CLI is a program. Each program file that you can execute under the operating system ends with the suffix .PR.

protocol A set of conventions defining the format for communication between programs; also the formal conventions established by official national committees to standardize networking practices (i.e., X.25, X.29, X.3, etc.).

pseudomacro A CLI construction, designed to make macros more useful, that expands to a current value. For example, !DATE returns the current date; !USERNAME returns the name of the user who invokes the macro.

queue A file that stores print and batch requests until the printer and system are ready to process them.

RDOS The Real-time Disk Operating System. A Data General operating system that can run two programs simultaneously.

record A series of one or more characters written to or read from a file. Records can be read and written by a text editor, in which case they are lines of text like these. Or, they can be read and written by your own application programs. AOS/VS recognizes four record types: data sensitive, fixed, variable, and dynamic. See the Sort/Merge chapter for details.

Release Notice A statement of recent software changes not documented in product manuals, that accompanies each software release. It's distributed on the same medium as the software.

remote Describes an item, such as a system, terminal, or database, that is managed by another computer, or by your computer over a communications line. The opposite of remote is local.

remote host A distant computer that does not manage your terminal, but one you can access through a network.

RMA The Resource Management Agent for the XODIAC network. RMA enables users and processes to access remote files, processes, and queues.

root directory The AOS/VS directory that contains and provides access to all other directories. Most often, a colon (:) represents the root directory.

SCOM A utility that compares source files line by line and compiles a list of file differences.

search list A list of directories that the CLI scans whenever it can't find the specified program or macro in the working directory. Usually the system manager sets the search list in a log-on macro, but users can set their own with the SEARCHLIST command.

secondary partition In RDOS, a fixed-length portion of the disk, roughly equivalent to an AOS/VS control point directory, that can contain files and subdirectories.

sector *See* disk block.

SED A screen-oriented text editor that runs under AOS and AOS/VS. See Chapter 4 for details.

software A set of computer programs, procedures, and possibly associated documentation concerned with the operation of a computer system — e.g., compilers, library routines, manuals, circuit diagrams.

Sort/Merge A utility program that will reorganize records within a file, combine several files into one, or translate EBCDIC files to ASCII.

source file A file designed for a specific purpose or application, consisting of original source code written by a programmer with a text editor. In BASIC, such a file can be brought into memory and run immediately. In a language such as FORTRAN or COBOL, a source file must be compiled and linked before it can be executed.

SPEED A character-oriented text editor that runs under AOS/VS, AOS, and RDOS. It allows you to execute commands conditionally and edit invisible characters.

stand-alone program A program that runs by itself, without an operating system to manage its I/O or supervise its scheduling.

stand-among program A version of a stand-alone program reconfigured to run under AOS/VS. The opposite of stand-alone.

streaming tape unit A tape unit that performs well (streams) when data arrives at a specific rate. If the data arives too slowly or too fast, the unit doesn't stream; therefore the read or write operation takes nearly 10 times as long.

subdirectory In RDOS, a variable length directory that can be created within a primary or secondary partition. A subdirectory grows and shrinks as files are added and deleted; it's similar to a subordinate directory in AOS/VS.

suffix Also called extension. A filename ending that defines the file's contents. For example, FORTRAN 77 source files end in .F77 and Pascal source files end in .PAS.

Superprocess An AOS/VS privilege that allows a user process to control any process on the system.

Superuser A privilege that allows a user to bypass file access controls (ACLs) and access any file on the system.

SWAT Data General's high-level, interactive symbolic debugging utility, which works with C, COBOL, FORTRAN 77, Pascal, and PL/I.

switch A slash (/) followed by a value. Switches modify the meaning of a command or the action performed on its arguments. For example, the /V switch on the DUMP command directs the CLI to verify the files copied to tape; DUMP with the switches /AFTER and /12-JAN-86 directs the CLI to copy only files created on or after January 12, 1986.

symbol A name that identifies some procedure, variable, array name, or location; it's often created by users, but sometimes defined by a language or system. For legal names, *see* name.

symbol table A file built by the Link utility to identify all symbols used in a program. Symbol tables are needed to debug programs or to patch program files.

synchronous line A communications line that uses a constant timing protocol, in which data transactions are controlled by a master clock and limited to a fixed period, to transmit or receive data. Synchronous lines are often used for high speed and/or long distance communications between computer systems. They are faster than asynchronous lines but require more system overhead.

system console The terminal connected directly to the CPU. User terminals are indirectly connected through a multiplexor that sorts incoming and outgoing data.

system manager The person who plans and administers an operating system; one of his or her tasks is deciding who will be allowed to use the system and what privileges they will have. Generally the system manager is responsible for backing up files and bringing the system up and down when necessary.

system operator *See* operator, system.

tape *See* magnetic tape.

template A symbol — any of the characters +, –, \, *, # — representing parts of pathnames in various ways (see Chapter 2 for details); also a cardboard or plastic strip that fits over or above the keyboard function keys (the top row of keys) and defines the key/command relationship for a given program.

terminal An interactive device with a keyboard for input and a screen for display. A hard-copy terminal writes on paper rather than a screen. The filename of any terminal is @CONSOLE.

text editor A program used to add text to a file or modify it. Typical capabilities of an editor include inserting, deleting, changing or reorganizing lines of text.

timesharing A computer system that's shared by several or many individuals and used for different purposes.

UDA Abbreviation for User Data Area. A storage area associated with a file, but not part of it. The UDA contains data for programs to read and act on. For example, a user, through the FCU utility, can insert print control characters in a UDA that specify the format of the printed file.

UDD Abbreviation of User Directory Directory. The system directory that contains the directory files of all valid users.

UNIX A large, multiuser operating system developed, along with the C programming language, by Bell Labs during the 1970s.

user data area *See* UDA.

user directory The directory created by the system manager and maintained by each user. It often becomes the working directory at logon; it can have subordinate directories that become at various times, working directories. *See also* UDD.

user profile A disk file, created by the system manager or someone else in authority, that defines the parameters of a person's account: the username, password, disk space allowance, and other privileges.

user, system Anyone who logs on an AOS/VS system. This can be a system manager or operator, a programmer or a general user.

username The name under which a system user logs on, and the name of the user directory; it's generally assigned by the system manager.

UTIL The utilities directory, containing most, if not all, utility programs on the system; its pathname :UTIL is often found on search lists.

utility; utility program A program supplied by Data General to help you develop your own programs; e.g., a text editor, compiler, linker, or debugger. Some utilities are shipped as part of the operating system; others are ordered separately.

virtual terminal An "invisible" terminal (with its own PID) that you use (via your own terminal) when you're logged on a remote host. VTA maintains the connection between your remote virtual terminal and your local physical terminal.

volume ID Also called VOLID. A six-character identifier the system writes on a labeled tape or diskette.

VTA The XODIAC network's Virtual Terminal Agent. VTA enables users to log on remote computer systems.

windowing Run by DG/VIEW, Data General's window management program, windowing allows you to run one or more processes in different areas on a terminal screen. DG/VIEW runs on Data General DS/7000-series computers.

word A sequence of digits treated as a single unit of information by the processor. In AOS/VS a word is 16 bits (2 bytes) long; a double word is 32 (4 bytes) bits long.

working directory The directory in which you are working (also called the current directory).

working set The number of 2,048-byte memory pages that a process is using, or has access to in memory. The PED utility displays the working-set size in column WSS.

X.25 An international communications protocol which Data General implements under AOS/VS in the XODIAC XTS process.

XLPT The name of the system process that manages printers and other spooled devices.

XODIAC A network management system consisting of software products that let interconnected systems exchange data and share resources. XODIAC offers several interfaces including the Virtual Terminal Agent (VTA), the File Transfer Agent (FTA), the Resource Management Agent (RMA), and the X.29/Host Packet Assembler/Disassembler (X.29/Host PAD).

End of Glossary

Index

Within this index, "f" or "ff" after a page number means "and the following page (or pages)". In addition, primary page references for each topic are listed first. Commands, calls, and acronyms are in uppercase letters (e.g., CREATE).

SYMBOLS

A

D

N

O

P

Q

T

Data General Corporation, Westboro, MA 01580

069-000031-02